

MATHEUS MARTINS JERONIMO

**DESENVOLVIMENTO DE ARQUITETURA DE
COMUNICAÇÃO DE UMA AERONAVE
MULTIRROTORA VIA *HARDWARE IN THE LOOP***



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA
2019

MATHEUS MARTINS JERONIMO

**DESENVOLVIMENTO DE ARQUITETURA DE
COMUNICAÇÃO DE UMA AERONAVE
MULTIRROTORA VIA HARDWARE IN THE LOOP**

Monografia apresentada ao Curso de Graduação em Engenharia Mecatrônica da Universidade Federal de Uberlândia, como parte dos requisitos para a obtenção do título de **BACHAREL EM ENGENHARIA MECATRÔNICA**.

Orientador: Prof. Felipe Machini Malachias Marques

Uberlândia - MG

2019

DEDICATÓRIA

Dedico este trabalho a minha vó, Abadia, e a minha mãe, Rosimeire, que sempre me auxiliaram e me guiaram nesta jornada. E, sobretudo, me ensinaram o que é o amor.

AGRADECIMENTOS

Gostaria de agradecer a Deus pela vida, bênçãos, amparo e oportunidade de realizar este sonho. Aos meus pais, Kleber e Rosimeire, especialmente a minha mãe pelo amor incondicional e por tantos ensinamentos e exemplos. À toda minha família pelo carinho, auxílio e acolhimento. À minha namorada, Aleixa, pelo afeto, companheirismo, amor e apoio.

Às amigas, Ana Paula e Renata, por toda ajuda durante a graduação, paciência e amizade. Aos amigos da Universidade que tornaram melhor a jornada da graduação.

Aos professores e mestres pelos ensinamentos e exemplos. À todas as pessoas que me ajudaram de forma direta e indireta na realização deste trabalho.

À Equipe de Desenvolvimento de Robótica Móvel (EDROM) pela oportunidade, pelos inúmeros aprendizados, pelas amizades e por todos os bons momentos. Ao Laboratório de Aeronaves Autônomas (LAA) pela oportunidade de participação e desenvolvimento deste trabalho, pelo conhecimento adquirido, aos amigos de trabalho membros do laboratório, em especial a Ivan Oliveira e Mateus Amarante, pela ajuda, ensinamentos e motivação.

Aos Prof. Dr. Leonardo Sanches, Prof. Dr. Roberto Mendes Finzi Neto e Prof. Felipe Machini Malachias Marques pela oportunidade, orientação, auxílio e inestimável apoio fundamentais para o desenvolvimento deste trabalho.

À Universidade Federal de Uberlândia e à Faculdade de Engenharia Mecânica pela oportunidade de realizar o curso de Engenharia Mecatrônica.

JERONIMO, M. M. **Desenvolvimento de Arquitetura de Comunicação de uma Aeronave Multirrotores via *Hardware in the Loop***. 2019. 148 f. Projeto de Fim de Curso, Universidade Federal de Uberlândia, Uberlândia, 2019.

RESUMO

As pesquisas relacionadas a técnicas de controle em conjunto com veículos aéreos não tripulados (VANTs) requerem metodologias que garantam segurança dos envolvidos e do veículo, e que possibilitem uma forma ágil de teste. Desta forma, é proposta a plataforma desenvolvida neste trabalho para atender estes requisitos. As finalidades do sistema são implementar a comunicação entre microcontrolador do VANT e um modelo dinâmico que representa a dinâmica da aeronave multirrotores, realizar simulação baseada em técnica *hardware-in-the-loop* (HIL) utilizando lei de controle do tipo LQR (Regulador Linear Quadrático) implementada no microcontrolador, e o modelo dinâmico. Como forma adicional de análise, são implementadas formas de acionar os atuadores do VANT durante a simulação. O sistema é desenvolvido através das linguagens de programação C e C++ no microcontrolador e através do *software* MATLAB®. Nas simulações, é proposto um controlador do tipo rastreador que permite a aeronave a seguir uma posição de operação desejada. A lei de controle é validada com base em simulações HIL. Também é avaliado o desempenho do sistema e verificado se o propósito do mesmo é alcançado.

Palavras Chave: Hardware-In-The-Loop, Veículo Aéreo Não Tripulado, Multirrotores.

JERONIMO, M. M. **Development of Communication Architecture of a Multirotor Aircraft via Hardware in the Loop**. 2019. 148 f. End of Course Project, Federal University of Uberlândia, Uberlândia, 2019.

ABSTRACT

The research related to control techniques in conjunction with unmanned aerial vehicles requires methodologies that guarantee safety of the involved and the vehicle, and that also make possible an agile form of test. In this way, it is proposed the developed platform in this work to meet these requirements. The purposes of the system are implement the communication between a UAV's micro-controller and a dynamical model that represents the dynamics of the multirotor aircraft, perform simulation based on hardware-in-the-loop (HIL) technique using LQR (Linear Quadratic Regulator) control law implemented in the microcontroller, and the dynamic model. As an additional form of analysis, ways of activating the UAV actuators during the simulation are implemented. The system is developed through C and C++ programming languages in the microcontroller and through the MATLAB® software. In the simulations, a tracker type controller is proposed which allows the aircraft to follow a desired operating position. The control law is validated based on HIL simulations. It also is evaluated the performance of the system and verified if the purpose of it is achieved.

Keywords: Hardware-In-The-Loop, Unmanned Aerial Vehicle, Multirotor.

Lista de Figuras

2.1	Sistema de coordenada <i>BCS</i> fixo no centro de gravidade da aeronave. Fonte: Fogelberg (2013)	5
2.2	Componentes do <i>powertrain</i> de motor <i>brushless</i> de um quadcóptero. Fonte: Conroy et al. (2014).	9
2.3	Indicação das matrizes de ganho no diagrama de bloco do sistema tipo Servomecanismo. Adaptação sobre a fonte: Lavretsky e Wise (2013).	14
3.1	Quadcóptero desenvolvido pelo Laboratório de Aeronaves Autônomas (LAA). Fonte: Marques (2018).	16
3.2	Vista lateral do VANT obtida em <i>software</i> CAD. Fonte: Marques (2018).	16
3.3	Fluxograma da interação entre os dois sistemas Servidor e Cliente.	20
4.1	Fluxograma da rotina principal implementada no MATLAB®.	22
4.2	Processo ilustrando a transformação das matrizes.	24
5.1	Fluxograma do código fonte principal implementado no microcontrolador.	35
5.2	Fluxograma dos processos para implementar o servidor no microcontrolador através da API de <i>socket</i>	46
5.3	Fluxograma dos procedimentos que ocorrem quando objeto da classe de comunicação é instanciado.	47
5.4	Fluxograma dos procedimentos que ocorrem internamente no momento de instanciar um objeto da classe LQR.	64
5.5	Ilustração do processo de construção de matriz com base em um vetor.	70
5.6	Fluxograma dos procedimentos realizados para estabelecer o estado da planta no controlador.	83

5.7	Fluxograma do processo de cálculo das saídas realizado pelo controlador.	88
5.8	Fluxograma dos procedimentos internos que ocorrem no método <i>setPWM()</i> da classe Motor.	97
5.9	Fluxograma que ilustra os processos que ocorrem para estabelecer velocidade nula nos motores.	102
5.10	Fluxograma dos processos que ocorrem internamente na função de emergência quando esta é invocada.	103
6.1	Velocidade angular dos motores 3 e 4 do VANT, em relação ao tempo, com mais detalhes.	111
6.2	Velocidade angular dos motores 1 e 2 do VANT, em relação ao tempo, com mais detalhes.	112
6.3	Posição do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.	113
6.4	Posição do CG do VANT, em relação ao tempo, com mais detalhes.	114
6.5	Velocidade linear do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.	115
6.6	Velocidade angular do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.	116
6.7	Ângulos de Euler referentes ao CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.	117
6.8	Tempo necessário para calcular estado da planta (solucionar ODE) e para comunicar com microcontrolador e computar lei de controle, submetida a entrada degrau unitário, em função do tempo.	118
6.9	Tempo necessário para calcular estado da planta (solucionar ODE) e para comunicar com microcontrolador e computar lei de controle, submetida a entrada degrau unitário, em função do tempo, com mais detalhes.	119

Lista de Tabelas

3.1	Propriedades mecânicas do quadcoptero. Fonte: Marques (2018)	16
3.2	Especificações técnicas do sistema de propulsão, ESC E300 DJI® e motor DC <i>brushless</i> 2212/920 KV DJI®. Fonte: Página <i>web</i> da DJI®	18

Lista de Símbolos

γ	- Ângulo entre o braço e eixo x_b
Θ	- Ângulos de Euler
ρ	- Densidade do ar
ϕ	- Ângulo de arfagem
θ	- Ângulo de rolagem
ψ	- Ângulo de guinada
Ω	- Velocidade angular do eixo do motor
A	- Matriz de estado
B	- Matriz de entrada
b	- Coeficiente de arrasto da hélice
g	- Aceleração devido a gravidade
J_m	- Momento de inércia da hélice em relação ao seu eixo de rotação
J_{xx}, J_{yy}, J_{zz}	- Momentos de inércia da aeronave
K	- Matriz de ganho
K_τ	- Constante proporcional do torque
K_t	- Constante relacionada ao torque produzido pelo motor
K_v	- Constante proporcional de voltagem
k	- Constante relativa a força de empuxo
k_d	- Coeficiente de arrasto da aeronave
l	- Distância entre atuador até o CG da aeronave
$\vec{M}$	- Vetor de momentos em relação ao centro de gravidade do corpo
m	- Massa
O_b	- Origem do sistema referencial do corpo
O_e	- Origem do sistema referencial inercial
P	- Velocidade angular em relação ao eixo x no sistema referencial do corpo
Q	- Velocidade linear em relação ao eixo y no sistema referencial do corpo
Q_k	- Matriz de ponderação de estado
R	- Velocidade linear em relação ao eixo z no sistema referencial do corpo

R_k	- Matriz de ponderação de controle
R_m	- Resistência elétrica do motor
R_B^I	- Matriz de rotação relacionando sistema referencial do corpo para o inercial
T	- Empuxo
u	- Velocidade linear na direção x no sistema referencial do corpo
$\vec{u}$	- Vetor das entradas de controle
V_x, V_y, V_z	- Velocidades lineares nas direções no sistema referencial inercial
v	- Velocidade linear na direção y no sistema referencial do corpo
$\vec{v}$	- Vetor velocidade linear nas direções do sistema referencial do corpo
w	- Velocidade linear na direção z no sistema referencial do corpo
$\vec{w}$	- Vetor velocidade angular referente ao sistema referencial do corpo
x_b, y_b, z_b	- Eixos/posição no sistema referencial do corpo
x_e, y_e, z_e	- Eixos/posição no sistema referencial inercial
$\vec{x}$	- Vetor de estado da planta
CG	- Centro de gravidade
BCS	- Sistema referencial do corpo
ICS	- Sistema referencial inercial

SUMÁRIO

1	INTRODUÇÃO	1
1.1	Objetivos	3
2	FUNDAMENTAÇÃO TEÓRICA	4
2.1	Modelo matemático dinâmico do multirroto	4
2.2	Lei de controle	10
3	SISTEMA	15
3.1	<i>Hardware</i> do VANT	15
3.2	Sistemas servidor e cliente	18
3.3	Conexão e interação entre os sistemas cliente MATLAB® e servidor microcontrolador	19
4	MATLAB®: rotina principal	21
4.1	Definir parâmetros iniciais de controle	22
4.2	Configurar parâmetros de rede	23
4.3	Adequar dimensões dos vetores dos parâmetros de controle	24
4.4	Conexão com o microcontrolador	26
4.5	Inicializar parâmetros do modelo do VANT	26
4.6	Enviar para o microcontrolador os parâmetros iniciais de controle	27
4.7	Enviar o número de iterações de simulação HIL para o microcontrolador	27
4.8	<i>Execução do laço de repetição</i>	27
4.9	Calcular estado da planta	28
4.10	Enviar estado da planta para o microcontrolador	28
4.11	Receber velocidades calculadas pela lei de controle	29

4.12	Atualizar o tempo e o estado da planta da iteração	29
4.13	Tempo de amostragem	30
4.14	Liberar recursos utilizados pelo objeto de comunicação	31
5	MICROCONTROLADOR	32
5.1	Configuração do PWM	33
5.2	Código fonte principal <i>main</i>	34
5.2.1	Instanciar objeto de comunicação	36
5.2.2	Instanciar objetos de motor	36
5.2.3	Iniciar função de emergência	37
5.2.4	Inicializar variáveis para armazenar parâmetros referentes a lei de controle e o número de iterações de simulação HIL	38
5.2.5	<i>while(true)</i>	38
5.2.6	Aguardar por conexão. Nova requisição de conexão	39
5.2.7	Receber parâmetros de configuração referentes a lei de controle	39
5.2.8	Receber número de iterações de simulação HIL	40
5.2.9	Imprimir na tela valores recebidos para verificação	40
5.2.10	Instanciar objeto do controlador LQR	40
5.2.11	Número de iterações	41
5.2.12	Receber estado da planta	41
5.2.13	Estabelecer o valor recebido de estado no controlador e calcular as ações de controle	42
5.2.14	Obter as velocidades dos motores calculadas pelo controlador	42
5.2.15	Converter as velocidades para valores de <i>duty cycle</i>	43
5.2.16	Definir os valores de <i>duty cycle</i> nos motores	43
5.2.17	Enviar os valores de velocidade para o MATLAB®	44
5.2.18	Encerrar conexão	44
5.2.19	Parar os motores	45
5.2.20	Fim do programa	45
5.3	Classe Comm	45

5.3.1	Instanciar objeto da classe de comunicação	47
5.3.2	Aceitar conexão com o cliente	54
5.3.3	Receber parâmetros	56
5.3.4	Imprimir dados recebidos	57
5.3.5	Receber estado da planta	59
5.3.6	Enviar dados da saída da lei de controle	59
5.3.7	Encerrar conexão com o cliente	61
5.4	Classe LQR	62
5.4.1	Instanciar objeto da classe LQR	63
5.4.2	Definir o estado da planta no controlador	82
5.4.3	Computar as saídas da lei de controle	87
5.4.4	Obter as saídas da lei de controle	91
5.4.5	Retornar dimensão, em <i>bytes</i> , do vetor das saídas do controlador	92
5.5	Classe Motor	93
5.5.1	Instanciar objeto da classe Motor	93
5.5.2	Converter velocidades em <i>duty cycles</i> equivalentes	95
5.5.3	Estabelecer valor de <i>duty cycle</i> nos motores	96
5.5.4	Encerrar rotação dos motores	101
5.6	Função de emergência <i>Emergency</i>	103
5.6.1	<code>paradaDeEmergencia()</code>	103
5.6.2	Declarar variáveis	104
5.6.3	Abrir os arquivos	104
5.6.4	Escrever nos arquivos	105
5.6.5	Verificar erro e fechar os arquivos	105
5.6.6	Informar o usuário	107
5.6.7	Encerrar execução do programa	108
6	RESULTADOS E DISCUSSÕES	109
6.1	Simulações	109
6.2	Resultados e análises	109

7	CONCLUSÕES E TRABALHOS FUTUROS	120
7.1	Conclusões	120
7.2	Trabalhos Futuros	121
8	REFERÊNCIAS BIBLIOGRÁFICAS	122
9	APÊNDICE	128
9.1	APÊNDICE A - Funções <i>Hexa</i> , <i>Forces</i> e <i>Moments</i>	128

CAPÍTULO I

INTRODUÇÃO

De acordo com Jorge e Inamasu (2014) o termo “Veículos Aéreos Não Tripulados” (VANT) se refere a aeronaves autônomas, semiautônomas ou remotamente operadas que são projetadas para voar sem a presença de piloto humano e controlada de forma remota ou de forma autônoma. Segundo Machini (2016) este tipo de veículo aéreo multirrotor apresenta uma simplicidade de controle na qual ajusta-se apenas a velocidade de rotação de cada motor de forma independente para realizar o controle das condições de operações como altitude, atitude e velocidades. Os multirrotores são aeronaves que apresentam diferentes número de motores como tricóptero (3 rotores), quadcóptero (4 rotores), hexacóptero (6 rotores) e octópteros (8 rotores) e são da categoria dos VANTs. Estes apresentam diversas aplicações nas áreas de agricultura de precisão, sensoriamento remoto, fotografia, dentre outras.

Este tipo de aeronave, devido a custo benefício, campo de aplicações e simplicidade mecânica, dinâmica e de manutenção é cada vez mais objeto de estudo e pesquisas (Machini (2016) e Suiçmez (2014)). Desta forma, VANTs ultimamente são utilizadas como plataforma para o teste e desenvolvimento de diferentes estratégias de técnicas de controle (Machini et al., 2018)

Nas pesquisas relacionadas a VANTs, é empregado o uso da técnica *hardware-in-the-loop* (HIL) para realizar testes e desenvolvimento de estratégias de controle. Esta técnica é uma simulação que envolve *software* e *hardware* na qual os sinais do controlador são conectados em um sistema que emula o funcionamento do sistema real que se deseja testar (National Instruments,

2019). A técnica HIL é uma simulação em tempo real contendo um modelo matemático do VANT real e o atuador que de fato seria adotado (Machini et al., 2018). O modelo é composto por um conjunto de equações diferenciais que são capazes de representar a dinâmica da planta. Em um computador é implementado o modelo dinâmico do sistema e em um microcontrolador embarcado na aeronave é implementado a lei de controle que será utilizada. Nesta simulação, o computador e sistema embarcado interagem entre si de forma a emular o funcionamento da planta e controlador reais.

São muitos os benefícios de se utilizar a HIL, como principais pode-se destacar os aspectos de segurança e agilidade pois o modelo matemático é usado em seu lugar. Desta forma é proporcionado segurança tanto aos usuários quanto à planta, pois, em caso de problemas com a lei de controle, nem a planta nem o usuário do sistema sofrerão danos. Segundo Machini et al. (2018), esta técnica permite identificar mau funcionamento do *hardware* e *software* relacionados ao VANT, microcontrolador e sistema de controle. Também permite prevenir e evitar danos tanto aos usuários do sistema de teste quanto a própria aeronave. De acordo com (National Instruments, 2019) através do HIL é possível realizar diversos testes representando diversas situações sem se preocupar com os gastos de recursos e tempo que seriam demandados caso o teste utilizando somente os componentes reais fosse realizado. Outro aspecto resultante do uso do HIL é a validação de diferentes lei de controle através dos dados obtidos da simulação.

O arranjo a ser desenvolvido neste trabalho consiste em um sistema de comunicação de dados entre um microcontrolador e um multirrotor com um computador por meio de protocolos de comunicação. No computador simular-se-á a dinâmica da planta enviando o estado da mesma para o sistema embarcado onde está contida uma lei de controle. Então, a partir dos estados estimados da planta obtidos numericamente, calcula os sinais de atuação dos atuadores e os envia iterativamente para a planta virtual.

Como uma forma adicional de avaliação da lei de controle, serão implementados comandos no sistema embarcado responsáveis por acionar os propulsores de uma aeronave protótipo fixa em uma bancada de testes. De acordo com as saídas obtidas pela lei de controle, será possível visualizar fisicamente os efeitos da lei de controle sobre o protótipo de bancada.

1.1 Objetivos

Este trabalho tem como objetivo geral a elaboração de uma plataforma de *hardware/software* de comunicação entre microcontrolador BeagleBone Green e um modelo numérico implementado computacionalmente no *software* MATLAB[®].

Como objetivos específicos podem ser listados os seguintes pontos: descrever como é realizada a comunicação entre os dispositivos, proporcionar forma de teste de atuadores como um motor elétrico, configurar e preparar o sistema para que possa ser embarcado, realizar avaliação de tempos de amostragem em função do processamento do microcontrolador, validar o modelo dinâmico proposto para multicópteros e elaborar material que sirva de apoio para implementação de leis de controle.

CAPÍTULO II

FUNDAMENTAÇÃO TEÓRICA

2.1 Modelo matemático dinâmico do multirrotor

Neste Capítulo descrever-se-ão a modelagem dinâmica de uma aeronave multirrotora.

De acordo com Machini (2016) e Valavanis (2007) aplicando-se a segunda lei de Newton em corpos rígidos com 6 graus de liberdade (3 translações e 3 rotações) pode-se derivar as equações de movimento de um VANT multirrotor. Esta hipótese de corpo rígido é válida para este tipo de VANT e pode ser adotada (Machini, 2016).

São utilizados dois sistemas referenciais para descrever a aeronave, um sistema referencial do corpo definido como *Body Fixed Coordinate Frame (BCS)* e um referencial inercial da Terra, *Inertial Coordinate System (ICS)*. O sistema *BCS*, mostrado na Fig. 2.1, é fixo ao centro de gravidade (*CG*) do VANT e é indicado por $\{O_b; x_b, y_b, z_b\}$. Já *ICS* é fixo na superfície da Terra sendo indicado por $\{O_e; x_e, y_e, z_e\}$ (Marques, 2018). Deve ser destacado que para efeito de simplificação das equações do movimento, as propriedades de inércia e as variáveis cinemáticas, como velocidades lineares e angulares, da aeronave são descritas a partir do *BCS* (Machini, 2016). Para definir a orientação do multirrotor (*BCS*), são utilizados os ângulos de Euler, $\Theta = [\phi \ \theta \ \psi]^T$ que representam os ângulos de arfagem, rolagem e guinada, respectivamente, em relação ao sistema referencial inercial *ICS* (Machini, 2016).



Figura 2.1 – Sistema de coordenada *BCS* fixo no centro de gravidade da aeronave. Fonte: Fogelberg (2013)

As velocidades lineares do CG são definidas em relação aos eixos x_e , y_e e z_e de *ICS* como: $\vec{V}_e = [V_x \ V_y \ V_z]^T$, representando componentes da velocidade nestes eixos. Já as velocidades lineares e angulares são definidas em *BCS* como: $\vec{v} = [u \ v \ w]^T$ e $\vec{\omega} = [P \ Q \ R]^T$, respectivamente, sendo P, Q e R velocidades em relação aos eixos x_b , y_b e z_b . Sendo estas velocidades descritas em relação a um observador no sistema *ICS* como referencial (Marques, 2018).

Equações do movimento para força e momento

As equações do movimento em relação ao *BCS* são as forças e momentos externos aplicados no centro de massa da aeronave (Machini et al., 2018), isto é,

$$m \begin{bmatrix} \dot{u} - Rv + Qw \\ \dot{v} - Pw + Ru \\ \dot{w} - Qu + Pv \end{bmatrix} = \vec{F}_T^{BCS} + \vec{F}_D^{BCS} + \vec{F}_{grav}^{BCS} = \begin{bmatrix} -k_{dx}u + mg \sin \theta \\ -k_{dy}v - mg \sin \phi \cos \theta \\ \sum k\Omega_i^2 - k_{dz}w - mg \cos \phi \cos \theta \end{bmatrix} \quad (2.1)$$

$$\vec{M} = \begin{bmatrix} J_{xx}\dot{P} + QR(J_{zz} - J_{yy}) - J_{xz}(\dot{R} + PQ) + J_{yz}(R^2 + Q^2) + J_{xy}(\dot{Q} + PR) \\ J_{yy}\dot{Q} + PR(J_{xx} - J_{zz}) + J_{xz}(P^2 + Q^2) + J_{xy}(\dot{P} + RQ) + J_{yz}(\dot{R} + PQ) \\ J_{zz}\dot{R} + PQ(J_{yy} - J_{xx}) + J_{xz}(QR + \dot{P}) + J_{yz}(\dot{Q} - PR) + J_{xy}(Q^2 - R^2) \end{bmatrix} = \vec{\tau}_T + \vec{\tau}_G + \vec{\tau}_F$$

$$\vec{\tau}_T + \vec{\tau}_G + \vec{\tau}_F = \begin{bmatrix} (\sum l \sin(\gamma) k \Omega_i^2) + (\sum Q J_m \Omega_i) \\ (\sum -l \cos(\gamma) k \Omega_i^2) + (\sum P J_m \Omega_i) \\ \sum -\text{sign}(\Omega_i) b \Omega_i^2 \end{bmatrix} \quad (2.2)$$

A Eq. (2.1) se refere as forças externas, enquanto a Eq. (2.2), aos momentos externos.

Sobre a Eq. (2.1), as forças externas aplicadas na aeronave são constituídas pela força de empuxo ($\vec{F}_T^{BCS}$), pela força de arrasto aerodinâmico ($\vec{F}_D^{BCS}$) e pela força peso do VANT ($\vec{F}_{grav}^{BCS}$) (Marques, 2018). O termo k_d representa o coeficiente de arrasto da aeronave, onde k_{dx} , k_{dy} e k_{dz} representam suas componentes nos eixos x , y e z , respectivamente. A constante k é um fator relativo a força de empuxo (T). A variável Ω_i se refere a velocidade angular de um dado motor i . A constante m representa a massa da aeronave. O termo g é a aceleração da gravidade (Marques, 2018).

Em relação a Eq. (2.2), os momentos externos são compostos por torque devido a variação da força de empuxo ($\vec{\tau}_T$), torque em consequência do efeito giroscópio ($\vec{\tau}_G$) e torque em virtude do arrasto da hélice ($\vec{\tau}_F$) (Marques, 2018). O termo l refere-se ao comprimento do braço do multirrotor. Já o termo γ corresponde ao ângulo entre o braço e o eixo x_b relativo ao CG da aeronave. A constante k é um fator relativo a força de empuxo (T). A variável Ω_i se refere a velocidade angular de um dado motor i . O termo J_m é o momento de inércia da hélice com relação ao seu eixo de rotação. A função sign é utilizada para determinar o sentido do torque: anti-horário ou horário. O termo b representa um coeficiente de arrasto da hélice (Marques, 2018).

Equação de velocidade angular

De acordo com Machini (2016) a atitude do VANT é definida pelos ângulos de Euler representado por $\Theta = [\phi \ \theta \ \psi]^\top$, em que ϕ , θ e ψ são os ângulos de arfagem, rolagem e guinada, respectivamente. Então a velocidade angular pode ser definida como $\dot{\Theta} = [\dot{\phi} \ \dot{\theta} \ \dot{\psi}]^\top$. Deve-se destacar que $\omega \neq \dot{\Theta}$ pois enquanto o primeiro aponta na direção do eixo de rotação o segundo termo representa somente a derivada temporal dos ângulos de Euler. A seguir é apresentada a relação existente entre estes dois termos (Roskam, 2001):

$$\omega = \begin{bmatrix} 1 & 0 & -\sin \theta \\ 0 & \cos \phi & \cos \phi \cos \theta \\ 0 & -\sin \phi & \cos \theta \cos \phi \end{bmatrix} \dot{\Theta} \quad (2.3)$$

Os dois sistemas de eixos *ICS* e *BCS* podem ser relacionados também por meio dos ângulos de Euler utilizando-se matriz rotacional obtida em uma determinada sequência de rotações: guinada ($R_B^I(\psi)$), arfagem ($R_B^I(\theta)$) e rolagem ($R_B^I(\phi)$) (Machini, 2016)

$$R_B^I(\psi) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

$$R_B^I(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \quad (2.5)$$

$$R_B^I(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix} \quad (2.6)$$

De acordo com Machini (2016) a matriz de rotação R_B^I é obtida ao multiplicar os termos das Eqs. (2.4), (2.5) e (2.6) obtendo-se:

$$R_B^I = \begin{bmatrix} \cos \theta \cos \psi & \cos \psi \sin \phi \sin \theta - \cos \phi \sin \psi & \cos \psi \cos \phi \sin \theta + \sin \phi \sin \psi \\ \cos \theta \sin \psi & \sin \theta \sin \psi \sin \phi + \cos \phi \cos \psi & \cos \phi \sin \psi \sin \theta - \cos \psi \sin \phi \\ -\sin \theta & \sin \phi \cos \theta & \cos \theta \cos \psi \end{bmatrix} \quad (2.7)$$

A matriz R_B^I é ortogonal, isto é, $R_B^{I^{-1}} = R_B^{I^T}$. Logo pode-se transformar de *ICS* para *BCS* da seguinte forma (Machini, 2016):

$$\vec{q}_B = R_B^I \cdot \vec{q}_I = (R_B^I)^{-1} \cdot \vec{q}_I = (R_B^I)^T \cdot \vec{q}_I \quad (2.8)$$

Os vetores $\vec{q}_I$ e $\vec{q}_B$ representam quaisquer vetores no sistema *ICS* e *BCS* respectivamente (Machini, 2016).

Forças de propulsão

É comum o uso em conjunto de rotores e hélices como conjunto propulsor de aeronaves autônomas de pequeno porte. Pode-se dizer que, em especial, motores *brushless* são amplamente utilizados em aeronaves com massa superior a 100 gramas e motores *brushed* são mais utilizados em veículos menores (Conroy et al., 2014).

O sistema de propulsão elétrico geralmente utilizado em VANTs é constituído por um dispositivo controlador de velocidade eletrônico, mais conhecido como *eletronic speed controller* (ESC), rotor e hélice. Este sistema é comandado por um microcontrolador capaz de produzir um sinal de saída modularizado por largura de pulso (*pulse width modulation* - PWM) que é entrada para o ESC que é responsável por controlar a comutação do motor *brushless*. O valor de entrada dita a corrente elétrica média que o ESC fornecerá ao motor. Este por sua vez, utiliza esta potência elétrica para rotacionar seu eixo, rotacionando, assim, a hélice acoplada neste em uma determinada velocidade desejada. Devido à aerodinâmica da hélice, será produzido um vetor empuxo normal ao raio da hélice resultando, desta forma, na sustentação a aeronave (Marques, 2018). O *powertrain* do VANT é mostrado na Fig. 2.2.

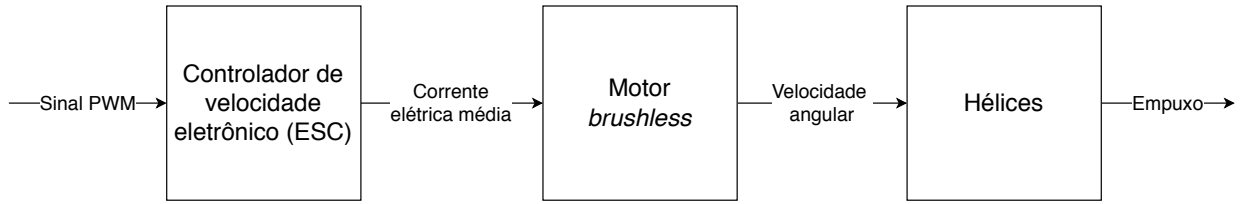


Figura 2.2 – Componentes do *powertrain* de motor *brushless* de um quadróptero. Fonte: Conroy et al. (2014).

De acordo com Marques (2018), o empuxo (T) produzido pela hélice pode ser calculado da seguinte forma:

$$T = \left(\frac{K_\tau K_v \sqrt{2\rho\pi R_m^2}}{K_t} \Omega \right)^2 = k\Omega^2 \quad (2.9)$$

Onde K_τ é uma constante de torque que relaciona o torque produzido com a força de empuxo, K_v é uma constante de voltagem que relaciona a voltagem aplicada e a velocidade angular do eixo do motor, K_t é uma constante relacionada ao torque gerado eletricamente pelo motor relacionando o torque e a corrente de entrada do motor, ρ é a densidade do ar, R_m é a resistência elétrica do motor. O termo Ω é a velocidade angular do motor. A constante k relaciona a força de empuxo e a velocidade do motor (Marques, 2018).

Através da Eq. (2.9) pode-se concluir que a força de empuxo produzida por um rotor é proporcional ao quadrado da velocidade angular (Ω) e a uma constante (k). Para o caso de um VANT de classe quadróptero o empuxo total pode ser calculado da seguinte maneira (Marques, 2018):

$$T_{total} = \sum_{i=1}^4 T_i = \begin{bmatrix} 0 \\ 0 \\ \sum k\Omega_i^2 \end{bmatrix} \quad (2.10)$$

2.2 Lei de controle

Para que possam ser aplicadas técnicas modernas de controle, como por exemplo, a técnica LQR (*Linear Quadratic Regulator*), as equações devem ser linearizadas (Burns, 2001).

Após efetuar o procedimento de linearização, o sistema pode ser descrito no espaço de estados na seguinte forma (Marques, 2018):

$$\dot{\vec{x}} = A\vec{x} + B\vec{u} \quad (2.11)$$

Conforme pode-se observar na Eq. (2.11), o vetor de estado da planta é representado por $\vec{x}$. Neste, está contido 12 coordenadas da aeronave como as 3 posições espaciais, 3 posições angulares, 3 velocidades de translação e 3 velocidades angulares, todas relacionadas ao sistema de coordenadas *BCS* e os ângulos de Euler (Marques, 2018):

$$\vec{x} = \{u \quad v \quad w \quad P \quad Q \quad R \quad \phi \quad \theta \quad \psi \quad x_e \quad y_e \quad z_e\} \quad (2.12)$$

Onde o termo $\vec{u}$, da Eq. (2.11), representa as entradas ou variáveis de controle (Machini, 2016). Já os termos A e B, segundo Marques (2018), são matrizes definidas da seguinte forma:

[illegible]

$$B = \begin{bmatrix} 0 & \frac{k \sin(\gamma_i) k \Omega_{0_i}^2}{m} & \frac{k \cos(\gamma_i) k \Omega_{0_i}^2}{m} \\ 0 & -\frac{k \cos(\gamma_i) k \Omega_{0_i}^2}{m} & \frac{k \sin(\gamma_i) k \Omega_{0_i}^2}{m} \\ 2k\Omega_{0_i} & 0 & 0 \\ \frac{2l_i \sin(\gamma_i) k \Omega_{0_i}}{J_{xx}} & -\frac{b \sin(\gamma_i) \Omega_{0_i}^2}{J_{xx}} & -\frac{b \cos(\gamma_i) \Omega_{0_i}^2}{J_{xx}} \\ \frac{2l_i \cos(\gamma_i) k \Omega_{0_i}}{J_{yy}} & \frac{b \cos(\gamma_i) \Omega_{0_i}^2}{J_{yy}} & -\frac{b \sin(\gamma_i) \Omega_{0_i}^2}{J_{xx}} \\ -\frac{2b\Omega_{0_i}}{J_{zz}} & \frac{-(\sin(\gamma_i) + l_i \cos(\gamma_i)) l_i k \Omega_{0_i}^2}{J_{zz}} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.14)$$

De acordo com Marques (2018) após a linearização e a descrição do sistema na forma de espaço de estado conforme Eqs. (2.12), (2.13) e (2.14), técnicas de controle moderno podem ser aplicadas para capacitar o multirroto a seguir determinada trajetória desejada.

LQR regulador quadrático linear

A técnica de controle regulador quadrático linear (LQR) é uma das técnicas mais utilizadas na área espacial (Lavretsky e Wise, 2013). Esta técnica apresenta boa performance, robustez e minimização do uso de controle. Constituindo boa ferramenta de projeto em sistemas

lineares (Marques, 2018).

De acordo com Marques (2018) a lei de controle para sistemas lineares é definida como:

$$\vec{u}(t) = -K\vec{x}(t) \quad (2.15)$$

Sendo K a matriz de ganho. Esta matriz é determinada através da solução da equação algébrica de Riccati (Marques, 2018):

$$\tilde{P}A + A^T\tilde{P} + Q_k - \tilde{P}BR_k^{-1}B^T\tilde{P} = 0 \quad (2.16)$$

Em que Q_k e R_k representam as matrizes de ponderação de estado e de esforço de controle, respectivamente. A partir da Eq. (2.16) a matriz de ganho ótima do LQR é calculada da seguinte forma:

$$K = R_k^{-1}B^T\tilde{P} \quad (2.17)$$

Entretanto, o problema de LQR pode ser estendido para um rastreador de comando de forma que os ganhos K , representados na Fig. 2.3, são calculados da seguinte forma:

$$K_c = [K_p \quad K_{p-1} \quad \dots \quad K_1 \quad K_x] \quad (2.18)$$

Os termos $K_p \quad \dots \quad K_1$ são as matrizes das integrais de erro de controle e K_x é a matriz de estabilização da planta relativa as Eqs. (2.13) e (2.14) (Marques, 2018).

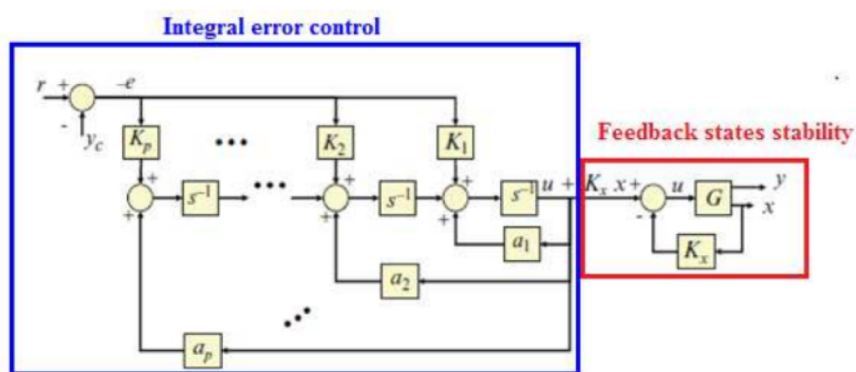


Figura 2.3 – Indicação das matrizes de ganho no diagrama de bloco do sistema tipo Servomecanismo. Adaptação sobre a fonte: Lavretsky e Wise (2013).

CAPÍTULO III

SISTEMA

Nesta Capítulo serão apresentados os *hardwares* do VANT, os componentes do sistema e a conexão entre estes.

3.1 *Hardware* do VANT

O VANT utilizado neste projeto trata-se de um quadcóptero, que pode ser visualizado na Fig. 3.1, desenvolvido pelo Laboratório de Aeronaves Autônomas (LAA) da Universidade Federal de Uberlândia (UFU). O mesmo é constituído por um microcontrolador, sistema de propulsão e fonte de energia de bancada. Este multirrotor foi projetado utilizando-se *software* CAD e construída placa de circuito impressa, *printed circuit board* (PCB), e filamento *acrilonitrila butadieno estireno* (ABS), sendo assim, pode-se utilizar técnica de impressão 3D para reproduzir o projeto. A aeronave apresenta as dimensões, em *mm*, informadas na Fig. 3.2.

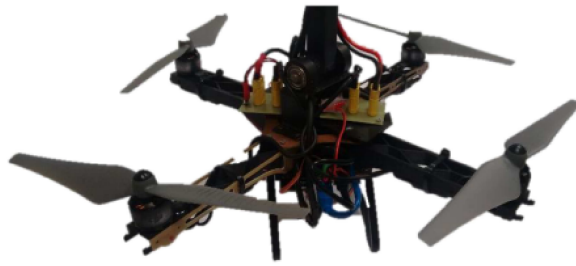


Figura 3.1 – Quadcóptero desenvolvido pelo Laboratório de Aeronaves Autônomas (LAA).

Fonte: Marques (2018).

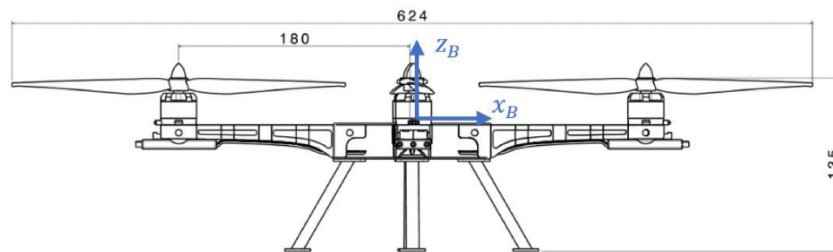


Figura 3.2 – Vista lateral do VANT obtida em *software* CAD. Fonte: Marques (2018).

As propriedades mecânicas da aeronave, são adquiridas por meio de uma balança e utilizando-se *software* CAD para estimá-las a partir do modelo. Estas propriedades são mostradas na Tab. 3.1:

Tabela 3.1 – Propriedades mecânicas do quadcoptero. Fonte: Marques (2018)

Propriedade	Parâmetro	Valor	Unidade [SI]
Número de rotores	n	4	—
Massa	m	0,468	kg
Momento de inércia em relação ao eixo x_b	J_{xx}	0,0049	$kg.m^2$
Momento de inércia em relação ao eixo y_b	J_{yy}	0,0049	$kg.m^2$
Momento de inércia em relação ao eixo z_b	J_{zz}	0,0088	$kg.m^2$
Coefficiente de arrasto	k_d	4,8e-03	$N.s^2$
Comprimento do braço	l	0,225	m

Pode-se listar como componentes do sistema: microcontrolador BeagleBone Green e o sistema de propulsão composto por quatro ESC E300 DJI®, motores DC *brushless* 2212/920 KV DJI®, sendo dois destes motores com sentido horário (CW) de rotação e os outros, em sentido anti-horário (CCW), e quatro hélices 9443 240 mm.

O microcontrolador apresenta as seguintes características¹:

- Processador Texas Instruments Sitara AM3358 1 GHz ARM® Cortex-A8
- 512 MB de memória RAM DDR3
- 4 GB de memória *flash* interna eMMC
- Capacidade de extensão de armazenamento através da entrada de cartão MicroSD
- Acelerador gráfico 3D
- 2 microcontroladores 32-bit com unidades de tempo real programável
- Porta USB para comunicação e fonte de alimentação
- Porta *Ethernet*
- 7 pinos analógicos de entrada e saída (I/O)
- 65 pinos digitais I/O
- 8 pinos de saída PWM
- Pinos com funcionalidades de *Timer*, UART, I2C e SPI.
- Compatibilidades com os sistemas operacionais Debian, *Android*, *Ubuntu* e outros.

Neste projeto o sistema operacional utilizado é a distribuição *Ubuntu* do Linux.

O sistema de propulsão utilizado neste projeto apresenta as seguintes características:

¹Disponível em: <<http://wiki.seeedstudio.com/BeagleBone.Green/>>. Acesso em: 10 maio 2019.

Tabela 3.2 – Especificações técnicas do sistema de propulsão, ESC E300 DJI® e motor DC

brushless 2212/920 KV DJI®. Fonte: Página *web* da DJI® ²

Componente	Parâmetro	Valor
Sistema de propulsão	Carga recomendada	300 g/eixo
	Máximo empuxo	600 g/eixo
ESC	Corrente	15 A
	Frequência do sinal	30 Hz ~450 Hz
	Voltagem	11,1 V ~14,8 V
Motor	Dimensão do estator	22 x 12 mm
	KV	920 rpm/V
	Massa	50 g
Rotor	Diâmetro / “ <i>Thread Pitch</i> ”	24 x 11 cm

3.2 Sistemas servidor e cliente

O sistema que implementa o HIL é composto por um servidor e um cliente. O código implementado no microcontrolador atua como o servidor e a rotina implementada no MATLAB® atua como cliente. O mesmo comunica-se com o microcontrolador através da rede local utilizando protocolo TCP/IP usado para garantir que os dados trocados não sejam perdidos, pois estes são cruciais para os cálculos das saídas das leis de controle e os estados da planta.

A implementação do sistema servidor é feita utilizando as linguagens C e C++ e o sistema cliente, utilizando-se software MATLAB®. Foram desenvolvidas classes que colocam em prática determinados aspectos do sistema de forma modularizada e independente. A classe Motor é responsável por prover funcionalidades relacionadas ao acionamento, conversão de unidades relacionadas e inicialização do motor no microcontrolador. A classe Comm realiza a comunicação entre BeagleBone Green e MATLAB®. LQR é responsável por executar a lei de controle baseada na técnica LQR. O método definido em Emergency disponibiliza função para ser acionada em caso de emergência.

²Disponível em: <<https://www.dji.com/br/e300/spec>>. Acesso em: 10 maio 2019.

3.3 Conexão e interação entre os sistemas cliente MATLAB® e servidor microcontrolador

A forma de estabelecer conexão entre a rotina no MATLAB® (cliente) e o microcontrolador (servidor) é a seguinte: em primeiro lugar, em um computador, denominado de computador 1, é realizada conexão com a BeagleBone Green através do protocolo Secure Shell (SSH). Protocolo de rede criptográfico, utilizado para realizar login neste sistema e, após o acesso, comandar o microcontrolador, de forma remota através de linha de comando, e transferir arquivos. Pode-se utilizar o protocolo de comunicação USB para acessar o microcontrolador, porém o protocolo SSH provê maior liberdade e capacidade para utilizar o sistema operacional da BeagleBone diferentemente do USB.

Para acessar o microcontrolador via SSH pode-se utilizar terminal do sistema operacional Linux utilizando o comando *ssh* do mesmo ou utilizando o *software PuTTY* instalado em um sistema operacional Windows®.

Após estabelecida a conexão, o computador 1 executa o código fonte principal, colocando o servidor em execução. Este aguardará por uma conexão de um cliente. Após o servidor entrar no estado de espera, em outro computador, denominado computador 2, será executado o cliente. Este irá se conectar com o servidor. A partir deste momento as interações entre os sistemas se inicia ocorrendo o *hardware-in-the-loop*. O fluxograma da Fig. 3.3 sintetiza o uso do sistema.

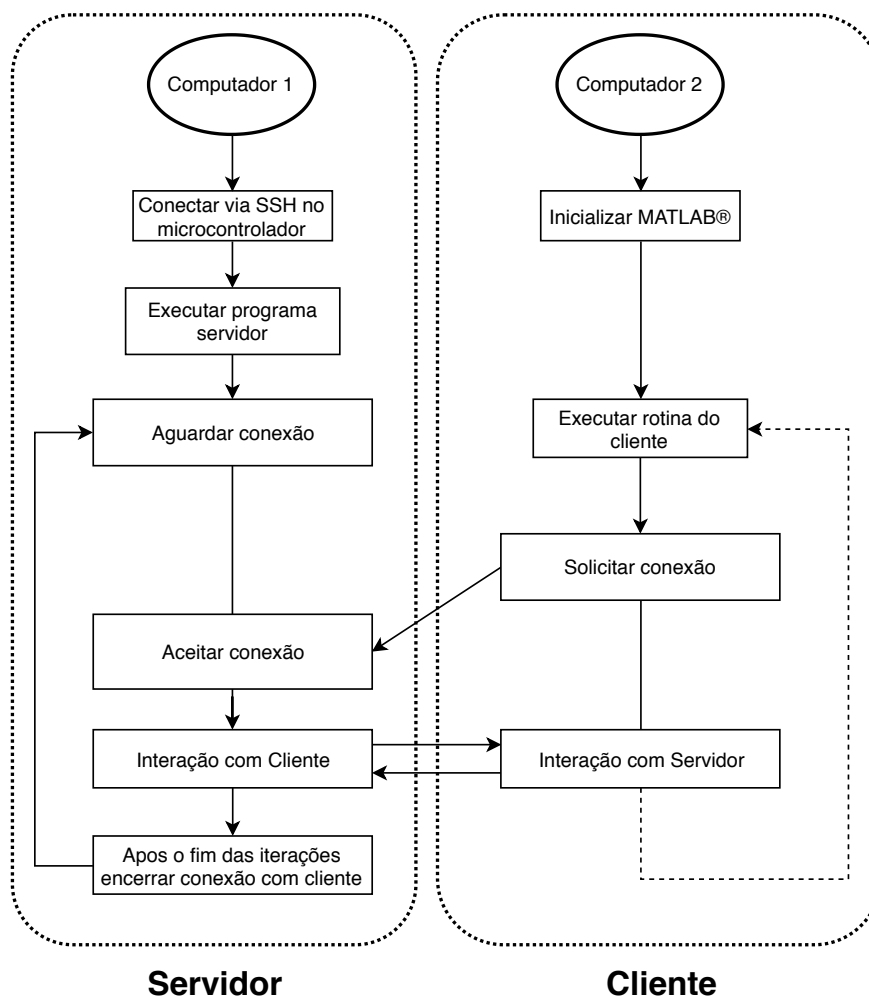


Figura 3.3 – Fluxograma da interação entre os dois sistemas Servidor e Cliente.

Segundo Fig. 3.3, quando a conexão é estabelecida haverá interações entre os sistemas até que determinado número de iterações seja atingido. Então o servidor encerra a conexão com o cliente. Caso o usuário queira uma nova interação, ele poderá se conectar novamente ao servidor que após o fim da conexão voltou para um estado de espera. A linha não contínua demonstra a opção de reinteração com o servidor.

Outro ponto que se pode observar no fluxograma é a linha continua conectando os blocos “Aguardar conexão” e “Aceitar conexão”. Não existe conexão direta entre esses blocos iguais as outras ao longo do fluxograma porque após realizar o procedimento “Aguardar conexão” só será realizado o próximo passo no servidor quando o cliente tiver solicitado conexão. Este processo pode ou não ocorrer de forma imediata no decorrer do tempo. A mesma lógica é valida para os procedimentos “Solicitar conexão” e “Interação com Servidor”.

CAPÍTULO IV

MATLAB®: rotina principal

Neste Capítulo será apresentada a rotina principal responsável por comunicar com a BeagleBone Green, enviar os parâmetros iniciais e de controle, calcular o estado da planta, enviar o estado da planta e receber as saídas do controlador. Em seguida serão mostradas as rotinas que simulam a planta.

A fim de tornar mais clara a metodologia da implementação do sistema, em vários fragmentos de códigos não serão mostrados todo o código a fim de explicitar o ponto a ser discutido. Estes fragmentos ocultados são apresentados como “//(...)” presente nos códigos desta Capítulo.

O fluxograma exibido na Fig. 4.1 resume as principais ações executadas na rotina que implementa parte do HIL no ambiente do MATLAB®.

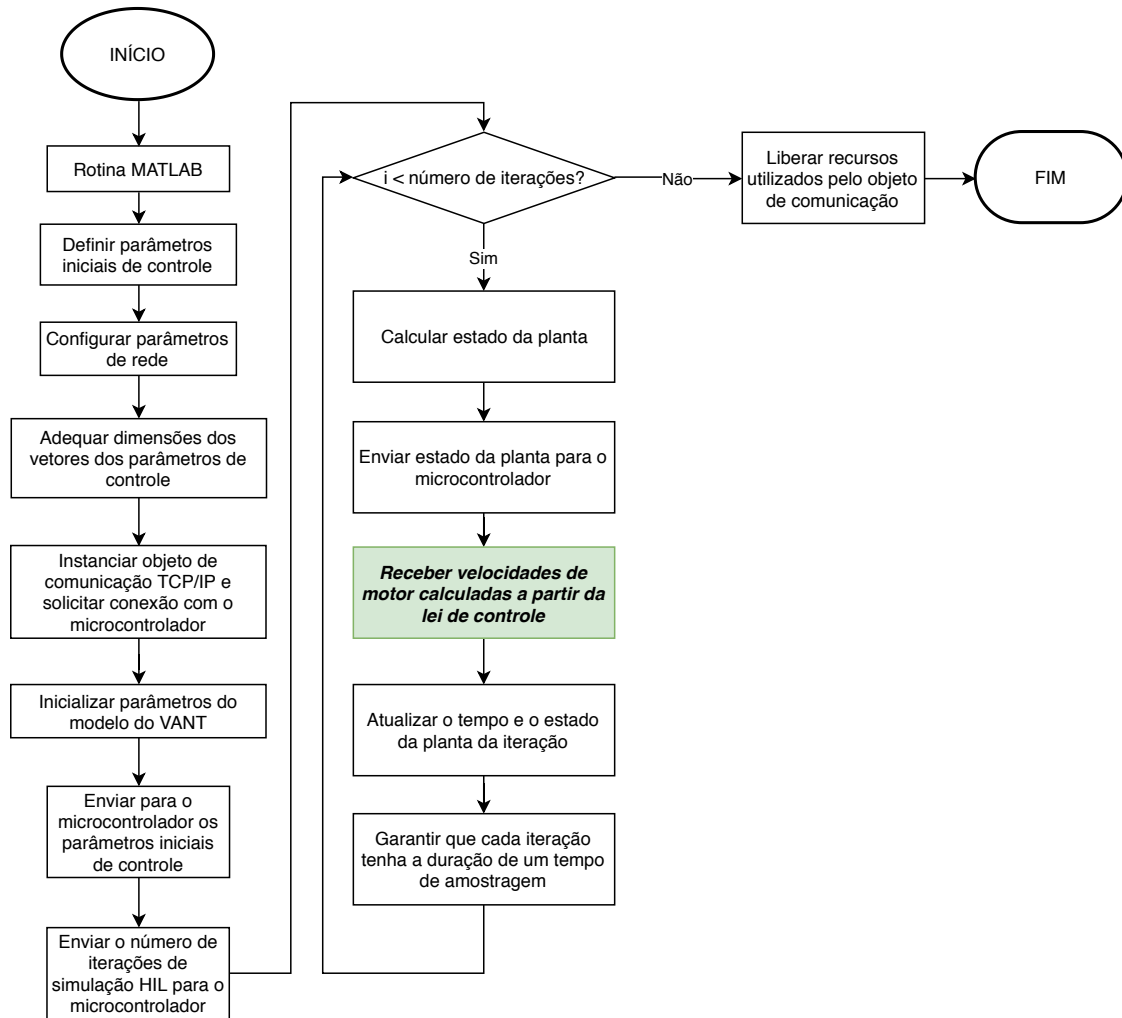


Figura 4.1 – Fluxograma da rotina principal implementada no MATLAB®.

4.1 Definir parâmetros iniciais de controle

Estes parâmetros são necessários para o controlador do tipo LQR. Sendo estes: o estado da planta (XX) que representa o vetor espaço de estado $\vec{x}$ conforme Eq. (2.12), velocidade dos rotores (ω), a condição de trimagem ($trim$) que possui ponto de operação de *hover* na qual todos os 12 elementos de $\vec{x}$ são nulos, tempo de amostragem (dt), a referência ($CoefRef$) que é implementada na forma de equações de primeiro grau de forma que a referência é representada como $A_x * t + B_x$, $A_y * t + B_y$ e $A_z * t + B_z$ onde t representa o tempo. Desta forma $CoefRef$ armazena os coeficientes angulares e lineares da reta relativos aos três eixos. Este valores podem ser alterados na rotina no MATLAB® e posteriormente serão enviados para o microcontrolador

sem que seja necessário recompilar o código na BeagleBone. Isso provê maior agilidade para realizar testes com diferentes parâmetros.

Neste bloco também são definidos o número de iterações, tempo inicial da iteração e tempo final.

```

1      % Definir parametros iniciais
2      % Parametros de controle
3      XX = zeros(1, 12);           % Inicializar variavel do estado a ...
                                   % ser enviado para BBG
4      load matrices.mat;          % Carregar as matrizes K1 e Kx do ...
                                   % arquivo .mat
5      omega = zeros(1, 4);        % Velocidade dos rotores
6      trim = [198.9429; -198.9429; 198.9429; -198.9429]; % Condicao de hover
7      dt = 0.06;                  % Tempo de amostragem [s]
8      CoefRef = [0 1; 0 1; 0 1];  % Coeficientes da referencia
9      condicoes_iniciais = zeros(size(XX));
10     omega = trim;                % Melhorar legibilidade do codigo
11
12     nIteracoes = 1000;           % Numero de iteracoes
13     t_int_inicial = 0;            % Tempo de iteracao inicial
14     t_int_final = dt;            % Tempo de iteracao final
15     xx_ = zeros(nIteracoes, 12); % Inicializar variavel para ...
                                   % armazenar os estados de cada iteracao
16     omega_ = zeros(nIteracoes, 4); % Inicializar variavel para ...
                                   % armazenar as velocidades de cada iteracao

```

4.2 Configurar parâmetros de rede

Nesta Seção é definido o IP do servidor com o qual irá se comunicar. *IP_BeagleBone* é usado para comunicar com o microcontrolador. *IP_Local* é usado para comunicar localmente entre processos executando no mesmo computador. O *IP_Local* é utilizado para realizar testes na comunicação, pois é possível executar código que simule o servidor no mesmo computador

que está sendo executado o processo do cliente, então pode-se testar se há falhas na rede ou no algoritmo de comunicação utilizado. *PORT* define a porta usada na comunicação.

São usadas variáveis para melhorar a legibilidade do código indicando o tipo de dado que será enviado e recebido (*DATA_TYPE*). Também é indicado a quantidade de dados que serão recebidos (*NUMBER_OF_DATA_2_RCV*). Neste caso são recebidos quatro dados pois no VANT são utilizados quatro rotores.

```

1      % Configuracao de rede - comunicacao
2      IP_BeagleBone = '10.0.33.236';
3      IP_Local = '127.0.0.1';
4      PORT = 8080;                % Porta definida de forma arbitrária
5      DATA_TYPE = 'double';      % Melhorar legibilidade do código
6      NUMBER_OF_DATA_2_RCV = 4;   % Melhorar legibilidade do código
7                                  % Obs: 4 pois são as velocidades ...
                                  dos quatro rotores

```

4.3 Adequar dimensões dos vetores dos parâmetros de controle

A função *write()* é responsável por enviar os dados do cliente (MATLAB®) para o servidor (BeagleBone). Segundo a implementação desta função no MATLAB®¹ os dados devem ser uma matriz de dimensão 1xN. Desta forma as matrizes a serem enviadas devem ser transformadas em um formato adequado para que possam ser enviadas. O processo para realizar este procedimento é ilustrado na Fig. 4.2.

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \Rightarrow [a_{1,1} \quad a_{2,1} \quad a_{3,1} \quad a_{1,2} \quad a_{2,2} \quad a_{3,2}]$$

Figura 4.2 – Processo ilustrando a transformação das matrizes.

¹Disponível em: <<https://www.mathworks.com/help/matlab/ref/tcpclient.write.html#buhajsp-2>>. Acesso em: 20 jul. 2018.

Ressalta-se que estruturas de dados como escalar ou vetor são enviados de forma direta sem serem transformadas por já estarem adequadas ao padrão $1 \times N$. O processo de adequar os dados ocorre no fragmento de código abaixo utilizando a sub-rotina *serialMatriz*:

```

1      % Adequar os parametros de controle
2      K1Vet = serialMatriz(K1);
3      KxVet = serialMatriz(Kx);
4      CoefRefVet = serialMatriz(CoefRef);
5      trimVet = serialMatriz(trim);

```

A implementação da sub-rotina é a seguinte:

```

1      function [serialVector] = serialMatriz(M)
2      % Adequa matriz (input) em vetor (1xN)
3
4      % Buffer para armazenar dimensao da matriz (linha e coluna)
5      lenVetMaux = size(M);
6
7      % Dimensao do vet. e num. de linha * num. de col. da matriz
8      lenVetm = lenVetMaux(1)*lenVetMaux(2);
9
10     % Vet. vazio onde sera armazenado a matriz transformada em vet.
11     serialVector = [];
12
13     for i=1:lenVetm
14         % A cada iteracao adiciona-se o elemento
15         % seguinte da matriz no final do vet.
16         serialVector = [serialVector M(i)];
17     end

```

4.4 Conexão com o microcontrolador

Para estabelecer a conexão com o microcontrolador deve-se instanciar objeto cliente TCP/IP, passando como parâmetros o IP do servidor e a porta. Após instanciar o objeto com sucesso a conexão está estabelecida entre MATLAB® e BeagleBone Green.

```

1      % Criar socket - Criar conexao TCP/IP com microcontrolador
2      t_obj = tcpclient(IP_BeagleBone, PORT)

```

4.5 Inicializar parâmetros do modelo do VANT

Os parâmetros dinâmicos da aeronave, conforme apresentados na Tabela 3.1, são definidas neste bloco:

```

1      % Inicializar parametros do modelo do VANT
2      %% Muticopter properties
3      param.n = 4;                                % Number of eletric motors
4      param.m = 0.468;                             % Multirrotoir mass
5      param.J = [0.0049,      0,      0;
6                  0, 0.0049,      0;
7                  0,      0, 0.0088];             % Inertia matrix
8      param.kd = 4.8e-3;                           % Drag coefficient
9      param.k = 2.9e-5;                             % Thrust coefficient
10     param.l = ones(1, param.n)*0.225;             % Multirrotoir arm length
11     param.gamma = [0, 90, 180, 270]*pi/180; % Multirrotoir arm angle wrt ...
        x axis
12     param.JJm = ones(1, param.n)*3.357e-5; % Propeller moment of inertia
13     param.b = 1.1e-6;                             % Propeller drag coefficient
14     param.zcg = ones(1, param.n)*0;               % Center of gravity z ...
        position BCS
15     param.g = 9.81;

```

4.6 Enviar para o microcontrolador os parâmetros iniciais de controle

Os parâmetros que tiveram suas dimensões adequadas e o escalar dt são enviados neste procedimento através da função `write()`. São atribuídos como parâmetros o objeto TCP/IP instanciado previamente e o dado que será enviado.

```
1      % Enviando parametros iniciais de controle para o microcontrolador
2      write(t_obj, K1Vet);
3      write(t_obj, KxVet);
4      write(t_obj, CoefRefVet);
5      write(t_obj, dt);
6      write(t_obj, trimVet);
```

4.7 Enviar o número de iterações de simulação HIL para o microcontrolador

A rotina cliente no MATLAB® determina a quantidade de iterações que ocorrerão na simulação. Este número, então, é enviado para o microcontrolador especificando no programa servidor o número de iterações.

```
1      % Enviando o numero de iteracoes de simulacao HIL
2      write(t_obj, nIteracoes);
```

4.8 Execução do laço de repetição

Após o envio das configurações iniciais, é executado o HIL no domínio da rotina do MATLAB®. Dentro deste laço ocorrerá um determinado número de iterações de simulação estabelecido. Nele, o estado da planta é calculado e enviado para o microcontrolador. A ação de controle é calculada e enviada de volta ao cliente. Então, essa é empregada na próxima iteração. Na execução do laço é medido o tempo gasto em uma iteração. Com base neste tempo

é realizado uma pausa de forma a garantir que uma iteração tenha duração equivalente ao tempo de amostragem dt .

4.9 Calcular estado da planta

O estado da planta é calculado através da integração numérica das equações de movimento ocorrendo a cada iteração. Utiliza-se para o mesmo, a função *ode15s* do MATLAB® que é aplicada para integrar sistema de equações diferenciais ordinárias do tipo *stiff* com valor inicial utilizando o método *variable order method*. Esta função possui precisão baixa a média. São atribuídos para o método, como entradas, definição de função anônima ($@(t, y)$) e a função a ser integrada (*Hexa*), o intervalo de integração ($[t_int_inicial \quad t_int_final]$), e, como último parâmetro, a condição inicial da função (*XX*). Destaca-se que como é realizada uma integração em cada iteração o intervalo de integração deve ser atualizado no fim de cada iteração.

Os resultados de saída de *ode15s* são pontos de avaliação (*T*) e a solução da função integrada (*XX*) que representa o novo estado da planta.

```
1      [T, XX] = ode15s(@(t, y) Hexa(y, param, omega, t), [t_int_inicial ...
      t_int_final], XX);
```

As equações do movimento baseadas no modelo de Newton-Euler para um corpo rígido de 6 graus de liberdade, descrita em (2.1) e (2.2), estão implementadas na função *Hexa*. Esta tem como entrada os parâmetros da aeronave, assim como as forças e momentos externos atuantes que dependem do vetor de estados e velocidade angular dos motores (fornecidos pelo microcontrolador) calculados numericamente a cada interação. Estas rotinas, desenvolvidas no LAA, *Hexa*, *Forces* e *Moments* serão apresentadas no Apêndice A.

4.10 Enviar estado da planta para o microcontrolador

Após realizar a evolução da dinâmica da planta, o estado atual é enviado para o microcontrolador utilizando-se a função *write()*.

```

1      % Enviar dado para BBG (estado da planta)
2      DATA_2_SEND = XX; % Melhorar legibilidade do código
3      write(t_obj, DATA_2_SEND);

```

4.11 Receber velocidades calculadas pela lei de controle

Este bloco é destacado no fluxograma da Fig. 4.1 indicando que as velocidades não são calculadas dentro desta rotina. Pois são calculadas no microcontrolador e são enviadas através de protocolo TCP/IP para o MATLAB®.

Os argumentos da função *read()* são o objeto TCP/IP, o número de dados a ser recebido e o tipo de dado. O microcontrolador enviará quatro dados correspondentes as velocidades calculadas para os rotores onde o tipo de dado é definido como *double*.

Os dados são recebidos no cliente e armazenados em *omega*. Esta função retorna uma matriz de dimensão 1xN, como os dados enviados do microcontrolador para o MATLAB® estão na forma de um vetor de 4 posições, não será necessário realizar conversão deste pois o cliente o interpretará como uma matriz 1x4.

```

1      % Receber dado da BBG (variavel de saída de controle)
2      omega = read(t_obj, NUMBER_OF_DATA_2_RCV, DATA_TYPE);

```

4.12 Atualizar o tempo e o estado da planta da iteração

No cálculo do estado da planta é realizada integração de equações diferenciais utilizando a função *ode15s*. A integração leva em conta o estado atual da planta e o intervalo de tempo que é o tempo de amostragem. A cada iteração o estado da planta e o intervalo de tempo devem ser atualizados:

```

1      condicoes_iniciais = XX;
2      t_int_inicial = t_int_final;
3      t_int_final = t_int_final + dt;

```

4.13 Tempo de amostragem

O tempo gasto para que ocorra a comunicação e o processamento da lei de controle é variável e é menor que o tempo de amostragem estabelecido de 60 ms. Dessa forma, faz-se o cliente esperar o tempo de amostragem para garantir que o tempo gasto das iterações seja constante.

O tempo de amostragem é definido de acordo com o tempo necessário para que o *solver* do MATLAB® determine a solução da *ode15s*. O tempo é escolhido de forma que seja superior ao tempo gasto para calcular o estado da planta, realizar comunicação e calcular as ações de controle. Porém, deve-se destacar, que a pausa realizada compromete a discretização do sistema.

Cada iteração segue o seguinte procedimento:

```

1      % Para medir a duracao da iteracao
2      tStart = tic;
3
4      %(...)
5      % Calculo do estado da planta
6      % Envio do estado
7      % Recebimento das saidas do controlador
8      %(...)
9
10     % Medir o tempo utilizado na iteracao
11     tempo_loop = toc(tStart);
12
13     % Armazenar os tempos gastos em cada iteracao
14     vetor_tempo_loop(i) = tempo_loop;

```

```
15  
16     % Garantir que a iteracao dure um tempo de amostragem  
17     pause(dt - tempo_loop);
```

Finalizada a iteração é checado se deve ser realizada outra iteração. Caso sim, estes procedimentos acima serão repetidos. Caso contrário, o procedimento “Liberar recursos utilizados pelo objeto de comunicação” será realizado.

4.14 Liberar recursos utilizados pelo objeto de comunicação

Neste ponto, o servidor encerrou a conexão com o cliente. Isso ocorre após finalizada a execução de todas iterações. Então, liberam-se os recursos utilizados nesta comunicação. Desta forma, caso o usuário deseje realizar novo processo de HIL o sistema poderá se conectar novamente com o servidor.

```
1     % Encerrando conexao e limpando recursos utilizados  
2     % relacionado ao objeto TCP/IP  
3     clear t_obj
```

Para analisar os resultados obtidos pode-se utilizar as variáveis que armazenam os dados de cada *loop*, como por exemplo, *xx_* que armazena os estados da planta e *omega_* que armazena as velocidades dos rotores.

CAPÍTULO V

MICROCONTROLADOR

Neste Capítulo serão apresentados os aspectos relacionados ao microcontrolador BeagleBone Green, sendo estes a configuração do PWM e o código fonte principal responsável por coordenar as funcionalidades do sistema HIL no lado do microcontrolador. Também serão descritas as classes e uma função de emergência utilizadas.

A fim de tornar mais clara a metodologia da implementação do sistema, em vários fragmentos de códigos não serão mostrados todo o código a fim de explicitar o ponto a ser discutido. Estes fragmentos ocultos são apresentados como “//(...)” presente nos códigos desta Capítulo.

As classes desenvolvidas são módulos que proporcionam funcionalidades ao sistema como realizar comunicação entre os dispositivos (classe Comm), implementar a lei de controle (classe LQR) e manipular os motores (classe Motor).

A função de emergência desenvolvida provê mecanismo para encerrar a rotação dos motores ao ser acionada pelo usuário do sistema numa situação de perigo.

O sistema foi criado de forma modularizada, isto é existem classes que são responsáveis por tarefas específicas e que possuem certa independência entre si. Para relacionar e gerenciar estas classes é utilizado o código fonte principal controlando, desta forma, o sistema.

A classe de comunicação é responsável por configurar a rede, enviar e receber dados. Já na classe de controle calculam-se as saídas com base na técnica LQR. A classe motor lida com os rotores. Por fim, a função de emergência lida exclusivamente com os recursos necessários para

parar o sistema, caso seja necessário.

5.1 Configuração do PWM

Os principais parâmetros que devem ser configurados são o período e o ciclo de trabalho (*duty cycle*). Estes são definidos através de arquivos que armazenam os valores destes parâmetros. Estes arquivos estão no diretório “pwm $\mathbf{Y}$ ” localizado no caminho “/sys/class/pwm/pwmchip $\mathbf{X}$ /” no sistema operacional do microcontrolador onde $\mathbf{X}$ e $\mathbf{Y}$ são números que mapeiam um determinado pino da BeagleBone.

Neste trabalho os pinos configurados como saída PWM no microcontrolador são os quatro pinos P9_14, P9_21, P9_22 e P9_42. Para configurar o PWM para cada um destes pinos utilizam-se os arquivos na seguinte relação:

- Pino: P9_14

Arquivos:

Duty cycle: “/sys/class/pwm/pwmchip3/pwm0/duty_cycle”

Período: “/sys/class/pwm/pwmchip3/pwm0/period”

- Pino: P9_21

Arquivos:

Duty cycle: “/sys/class/pwm/pwmchip1/pwm1/duty_cycle”

Período: “/sys/class/pwm/pwmchip1/pwm1/period”

- Pino: P9_22

Arquivos:

Duty cycle: “/sys/class/pwm/pwmchip1/pwm0/duty_cycle”

Período: “/sys/class/pwm/pwmchip1/pwm0/period”

- Pino: P9_42

Arquivos:

Duty cycle: `"/sys/class/pwm/pwmchip0/pwm0/duty_cycle"`

Período: `"/sys/class/pwm/pwmchip0/pwm0/period"`

O período do sinal e o *duty cycle* do PWM são configurados em nanosegundos na BeagleBone¹. Tecnicamente o valor do *duty cycle* é dado em porcentagem. Porém, no sistema operacional do microcontrolador, este parâmetro é configurado em unidade de tempo e é definido como o tempo em que o pulso está em nível lógico alto durante um período². Note que o tempo em alto deve sempre ser menor que o período. Desta forma, ao longo deste documento o termo *duty cycle* será utilizado de acordo a definição adotado no sistema operacional.

O período utilizado neste trabalho é 3 ms. Para configurar este parâmetro no microcontrolador deve-se escrever no arquivo "*period*" localizado em `"/sys/class/pwm/pwmchipX/pwmY/"` o valor 3000000. Já o *duty cycle* é estabelecido inicialmente como 1 ms. Para realizar a configuração deste parâmetro deve-se escrever no arquivo "*duty_cycle*", localizado no mesmo diretório que "*period*" o valor 1000000. Este valor é inicialmente adotado pois não produz rotação nos motores elétricos.

5.2 Código fonte principal *main*

O fluxograma exibido na Fig. 5.1 resume as principais ações executadas no código fonte que implementa parte do *hardware-in-the-loop* no ambiente do microcontrolador BeagleBone Green.

¹Disponível em: <http://processors.wiki.ti.com/index.php/Linux_Core_PWM_User%27s_Guide>. Acesso em: 21 jul. 2018.

²Disponível em: <<https://www.kernel.org/doc/Documentation/pwm.txt>>. Acesso em: 07 fev. 2019.

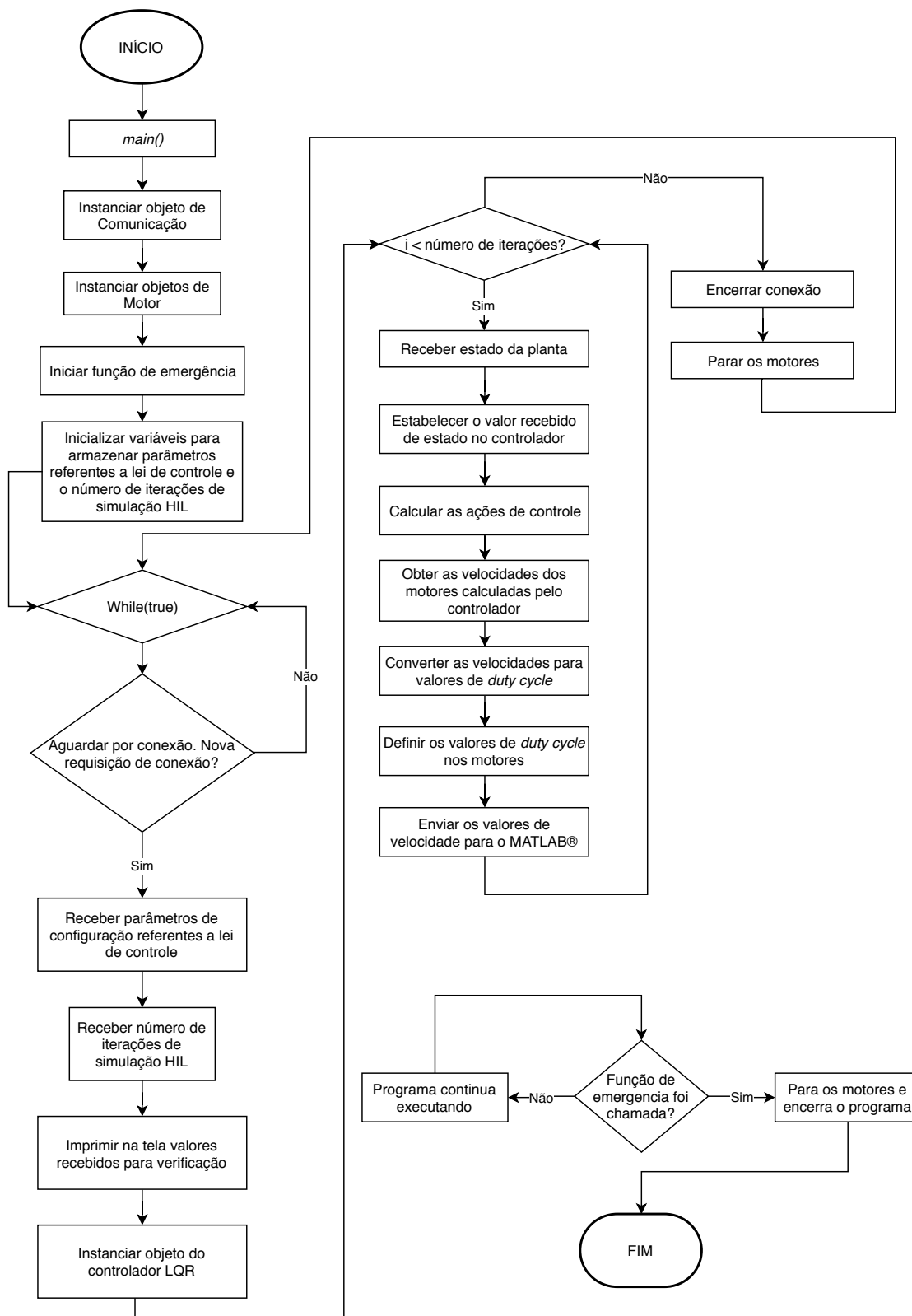


Figura 5.1 – Fluxograma do código fonte principal implementado no microcontrolador.

O sistema no lado do microcontrolador após realizar toda a configuração inicial entrará em um laço que não é previsto o seu fim. Este somente ocorrerá caso a função de emergência

seja invocada para que encerre a rotação dos motores imediatamente.

Este laço sem fim previsto provê maior agilidade no uso do sistema, uma vez que permite o programa no microcontrolador ser colocado em execução somente uma vez. Após iniciar o processo utilizando determinados parâmetros, o microcontrolador ficará aguardando uma nova conexão do MATLAB® contendo novos parâmetros para testes de acordo com o interesse do usuário. Sendo necessário somente alterar os mesmos na rotina no MATLAB® e executar a mesma sem ter a necessidade de toda vez ter que executar tanto o programa no microcontrolador quanto no MATLAB®.

5.2.1 Instanciar objeto de comunicação

No código principal o objeto é instanciado configurando uma conexão no modelo cliente/servidor, estabelecendo o microcontrolador como um servidor. Este aguardará conexão do MATLAB®.

De forma resumida, a classe configura os parametros de rede, estabelece o servidor do sistema, implementa a capacidade de enviar e receber dados e abrir e encerrar conexões. Posteriormente a classe Comm será detalhada.

```
1 Comm comm;
```

5.2.2 Instanciar objetos de motor

Os motores são representados por meio dos objetos da classe Motor, na qual funcionalidades como o acionamento são implementadas. Posteriormente a classe será detalhada.

Como parte da configuração inicial do sistema, são instanciados quatro objetos abstraindo cada motor do VANT. São atribuídos como argumentos para o construtor do objeto o sentido de rotação e o caminho onde está o arquivo que se configura o *duty cycle* do PWM de cada motor. O sentido de rotação é -1 ou +1 e seu valor depende do sentido de rotação da hélice conectada no eixo do rotor.

Os argumentos com prefixo *default* são diretivas definidas na classe. Elas são utilizadas para facilitar a legibilidade e manutenção do código. Dentre estes argumentos pode-se exemplificar *default_sentidoRotacaoM1* que é definido como -1 e *default_M1_path*, “/sys/class/pwm/pwmchip0/pwm0/duty_cycle”.

```
1      Motor motor1(default_sentidoRotacaoM1, default_M1_path),  
2          motor2(default_sentidoRotacaoM2, default_M2_path),  
3          motor3(default_sentidoRotacaoM3, default_M3_path),  
4          motor4(default_sentidoRotacaoM4, default_M4_path);
```

5.2.3 Iniciar função de emergência

A função de emergência é iniciada através da função *signal* da biblioteca *signal.h*. Esta é capaz de produzir interrupção de forma assíncrona no processo que a invoca em determinado evento. Neste trabalho, a interrupção ocorre quando as teclas *ctrl* e *c* são pressionadas simultaneamente. Isso invoca a função de emergência que para os rotores e encerra a execução do programa no servidor.

São atribuídos como argumentos para a função *signal()*: *SIGINT*, que representa um sinal de interrupção, e uma função que será executada quando o processo for interrompido que neste caso é a função de emergência: *paradaDeEmergencia()*

Em resumo, a função abre os arquivos de *duty cycle* de cada motor e escreve nestes um valor que produz como efeito o fim das rotações. Posteriormente a função de emergência será detalhada.

```
1      // Parada de emergencia - ctrl + c  
2      signal(SIGINT, paradaDeEmergencia);
```

5.2.4 Inicializar variáveis para armazenar parâmetros referentes a lei de controle e o número de iterações de simulação HIL

As variáveis são criadas como vetores destinados a armazenar os dados recebidos do MATLAB®. Tais vetores contêm as matrizes de ganho (*K1Vet*, *KxVet*), os coeficientes do polinômio de referência (*CoefRefVet*), a duração de uma iteração (*dtFromMatlab*), a trimagem (*trimVet*) e ao estado da planta (*XXVet*). Também cria-se variável para armazenar o número de iterações da simulação (*numeroIteracoes*). Por fim é criada variável que armazenará as saídas da lei de controle (*ptrOutCntToMatlab*), diferentemente das outras, os dados que serão armazenados nesta variável não são enviados pelo cliente, uma vez que, eles são calculados pela lei de controle implementada no microcontrolador.

A variável *ptrOutCntToMatlab* trata-se de um ponteiro que armazenará o endereço de memória do local onde a classe LQR armazena as saídas calculadas.

```

1  double K1Vet[iK1*jK1];
2  double KxVet[iKx*jKx];
3  double CoefRefVet[icoefRef*jcoefRef];
4  double dtFromMatlab;
5  double trimVet[iTrim*jTrim];
6  double XXVet[iSV*jSV];
7  double numeroIteracoes;
8
9  double *ptrOutCntToMatlab;
```

5.2.5 while(true)

Após as configurações iniciais do código fonte do microcontrolador é iniciado um laço que executará sem previsão para terminar. No início deste laço é verificado se existe solicitação de conexão. Caso não houver, o servidor ficará constantemente verificando se existe, de forma que quando ocorrer ele estará pronto para aceitar essa conexão.

Quando houver conexão será iniciado o HIL que ocorrerá durante um determinado

número de iterações pré-estabelecido. Quando este processo finalizar, a conexão com o cliente será encerrada e o servidor voltará ao estado inicial de verificar se há solicitação.

Vale notar que, apesar do laço não ter previsão para ser finalizado, pode-se terminar o processo por meio da função de emergência a qualquer momento.

5.2.6 Aguardar por conexão. Nova requisição de conexão

Após a configuração inicial do servidor, este é colocado em espera por nova conexão, quando houver a solicitação o servidor a aceitará e iniciará o *hardware-in-the-loop*. Quando não houver conexão pendente na fila do servidor, o mesmo continuará monitorando a fila em busca de nova solicitação.

```
1 // Beagle bone aceita conexao e estabelece conexao com o cliente
2 comm.accepting();
```

5.2.7 Receber parâmetros de configuração referentes a lei de controle

Após estabelecida a conexão, os primeiros dados são enviados para o microcontrolador. Estes são os parâmetros de controle que foram definidos no MATLAB® que o usuário deseja utilizar.

Como argumentos da função *receberDados()* são atribuídos o endereço da variável onde será armazenado o dado recebido e o tamanho, em *bytes*, do mesmo.

```
1 // Receber parametros iniciais do MATLAB
2 comm.receberDados(K1Vet, sizeof(K1Vet));
3 comm.receberDados(KxVet, sizeof(KxVet));
4 comm.receberDados(CoefRefVet, sizeof(CoefRefVet));
5 comm.receberDados(&dtFromMatlab, sizeof(dtFromMatlab));
6 comm.receberDados(trimVet, sizeof(trimVet));
```

5.2.8 Receber número de iterações de simulação HIL

Por fim é recebido o número de iterações relativo a simulação HIL.

```
1 comm.receberDados(&numeroIteracoes, sizeof(numeroIteracoes));
```

5.2.9 Imprimir na tela valores recebidos para verificação

Este procedimento tem como intuito proporcionar forma de visualização dos dados enviados para que possam ser verificados. Para realizar o mesmo, os dados recebidos são impressos na tela através da função *printData()* definida na classe de comunicação.

É atribuído como primeiro argumento da função uma *string* para ser impressa na tela. O segundo argumento é o endereço de memória do dado que se deseja imprimir. O último é o tamanho do dado, por exemplo, a quantidade de posições que o vetor possui.

```
1 comm.printData("Vetor K1 from matlab", K1Vet, iK1*jK1);
2 comm.printData("Vetor Kx from matlab", KxVet, iKx*jKx);
3 comm.printData("Vetor de Coeficientes de referencia from matlab", ...
    CoefRefVet, icoefRef*jcoefRef);
4 comm.printData("dt from matlab", &dtFromMatlab, 1);
5 comm.printData("Vetor trim from matlab", trimVet, iTrim*jTrim);
6 comm.printData("numeroIteracoes from matlab", &numeroIteracoes, 1);
```

5.2.10 Instanciar objeto do controlador LQR

O próximo procedimento é configurar o controlador LQR com base nos parâmetros recebidos. São atribuídos como argumentos do construtor da classe os parâmetros vetorizados referentes às matrizes de ganho, aos coeficientes do polinômio de referência, ao tempo de iteração (tempo de amostragem) e à trimagem. Desta forma é realizada a configuração interna. Após estes procedimentos o controlador está pronto para ser utilizado no código.

Os argumentos do construtor são os endereços dos vetores e no caso do *dtFromMatlab* é enviada uma cópia da variável em vez do endereço da mesma. Posteriormente a classe LQR que implementa o controlador baseado na técnica LQR no microcontrolador será detalhada.

```
1      LQR lqr(KlVet, KxVet, CoefRefVet, dtFromMatlab, trimVet);
```

5.2.11 Número de iterações

As iterações onde a ação de controle é computada ocorrem neste laço. Este *loop* será executado uma quantidade de vezes estabelecida pelo número de iterações no cliente, que neste trabalho é definido como 1000, porém o programador pode utilizar qualquer outro valor. Quando o mesmo tiver completado o número máximo de iterações, o servidor encerra a conexão com o MATLAB®, os motores são colocados em repouso e o microcontrolador retorna ao estado onde aguarda por conexões.

```
1      for(int i = 0; i < numeroIteracoes; i++)
2      {
3          // (...)
4      }
```

5.2.12 Receber estado da planta

Este procedimento será exercido quando houverem iterações para serem executadas. Caso contrário, o procedimento “Encerrar conexão” será efetuado.

O estado é calculado no programa cliente, conforme descrito em “Calcular estado da planta” no Capítulo 4. Este será armazenado no servidor através da função *receberDados()*. Os parâmetros desta são o endereço da variável onde será armazenado o estado e a quantidade de dados, em *bytes*, que será recebido.

```
1      comm.receberDados (XXVet, sizeof (XXVet));
```

5.2.13 Estabelecer o valor recebido de estado no controlador e calcular as ações de controle

Após o estado ser recebido no procedimento anterior, este é transmitido para o controlador através da função *setStateVariables()*. Atribuindo-se como parâmetros de entrada, o endereço do vetor referente ao estado da planta e a dimensão deste vetor. Este procedimento ocorre em cada iteração e, desta maneira, constantemente o novo estado da planta calculado no MATLAB® será entregue ao controlador.

Após o recebimento, o controlador realiza seus cálculos através da função *computeControl()*.

```
1      // Enviar estado para controlador
2      lqr.setStateVariables (XXVet, iSV*jSV);
3
4      // Calcular controle
5      lqr.computeControl();
```

5.2.14 Obter as velocidades dos motores calculadas pelo controlador

O controlador armazena as saídas em um determinado vetor interno membro da classe LQR. Neste trabalho, como são utilizados quatro motores, este vetor tem quatro posições para armazenar a velocidade calculada para cada motor. Estas saídas são obtidas no código fonte principal através do ponteiro *ptrOutCntToMatlab* que armazena o endereço do vetor e desta forma permite manipular estes dados no escopo do código.

```
1      // Obter saída do controlador
2      ptrOutCntToMatlab = lqr.getOutputControl();
3
```

```

4      /*
5      ptrOutCntToMatlab[0] corresponde ao motor 3
6      ptrOutCntToMatlab[1] corresponde ao motor 1
7      ptrOutCntToMatlab[2] corresponde ao motor 4
8      ptrOutCntToMatlab[3] corresponde ao motor 2
9      */

```

5.2.15 Converter as velocidades para valores de duty cycle

Neste trabalho, além da análise dos resultados do *hardware-in-the-loop* através de dados, como uma forma adicional de análise e inspeção visual, os motores são acionados durante a execução do sistema, variando suas velocidades de acordo com as saídas calculadas pela lei de controle, obtidas em rad/s. Para obtê-las nos motores é necessário que elas sejam convertidas para valores equivalentes em *duty cycle* relativos à técnica PWM. Este objetivo é alcançado através da função *convertVelocity_PWM()*. Para utilizá-la deve-se atribuir como argumento a velocidade a ser convertida. O resultado desta conversão é armazenado de forma interna em uma variável membra da classe Motor que está vinculada a uma determina instância desta classe.

```

1      motor1.convertVelocity_PWM(ptrOutCntToMatlab[1]);
2      motor2.convertVelocity_PWM(ptrOutCntToMatlab[3]);
3      motor3.convertVelocity_PWM(ptrOutCntToMatlab[0]);
4      motor4.convertVelocity_PWM(ptrOutCntToMatlab[2]);

```

5.2.16 Definir os valores de duty cycle nos motores

Neste procedimento, os valores armazenados internamente serão enviados aos arquivos de configuração de *duty cycle* dos motores para que sejam acionados. Para isso é utilizada a função *setPWM()*. Esta adquire estes dados, abre os arquivos e escreve nestes, desta maneira, acionando os motores.

Como observação pode-se dizer que a velocidade angular dos motores foi verificada utilizando-se tacômetro a fim de analisar se o comando de PWM resulta na velocidade desejada.

```

1      // Enviar dados de PWM para os motores
2      motor1.setPWM();
3      motor2.setPWM();
4      motor3.setPWM();
5      motor4.setPWM();

```

5.2.17 Enviar os valores de velocidade para o MATLAB®

Depois de escrever nos arquivos correspondentes a cada motor, são enviadas para o MATLAB®, utilizando-se a função *enviarDados()*, as velocidades, em rad/s, para que o novo estado da planta seja calculado. Deve-se atribuir como argumento para esta função o endereço da variável que armazena o dado e a quantidade de dados, em *bytes*, que serão enviados. Esta quantidade é obtida através da função *getOutputCtlSize()* que retorna o tamanho, em *bytes*, do vetor que armazena a saída do controlador na classe de controle.

```

1      comm.enviarDados(ptrOutCntToMatlab, lqr.getOutputCtlSize());

```

Após o envio para o cliente, caso existam mais iterações para serem realizadas o laço se repetirá. Caso não existam, o procedimento “Encerrar conexão” será executado.

5.2.18 Encerrar conexão

Quando finalizar a execução de todas iterações a conexão com o cliente é encerrada utilizando a função *closeSocket()*.

```

1      comm.closeSocket();

```

5.2.19 Parar os motores

Após o fim da conexão, são enviados comandos para encerrar o movimento dos motores. A função para realizar o mesmo é a *pararMotor()* sendo utilizada para cada instância de motor. No final destes procedimentos, é exibida uma mensagem na tela informando que os motores foram encerrados. Após a impressão, retorna-se ao ponto inicial do laço "*while(true)*".

```
1    motor1.pararMotor();
2    motor2.pararMotor();
3    motor3.pararMotor();
4    motor4.pararMotor();
5
6    std::cout << "Motores foram parados." << std::endl;
```

5.2.20 Fim do programa

O programa é encerrado quando o usuário utiliza a função de emergência ao detectar algum problema durante a execução usando as teclas *ctrl + c*. Quando a função é invocada ocorre a interrupção do processo. Esta é a única situação onde é prevista o fim do programa.

5.3 Classe Comm

Esta classe é responsável por realizar configuração e a implementação da comunicação entre os dois elementos do sistema: microcontrolador e MATLAB®. Para realizar a troca de pacote de dados entre os dois é usado uma interface de programação de aplicações, mais conhecida como API, de *sockets*. Esta interface implementa um programa servidor no microcontrolador e é através dela que a BeagleBone é capaz de receber dados do cliente e enviar dados para o mesmo. A obra de Coulouris, Dollimore e Kindberg (2007) fundamenta a implementação da classe de comunicação assim como o uso de *sockets* na classe.

O fluxograma da Fig. 5.2 indica os procedimentos para configurar o servidor, estabelecer a conexão com o cliente, realizar a troca de mensagens e encerrar conexão utilizando

os recursos da API de *socket*. Entretanto, neste trabalho, a comunicação entre os sistemas não ocorre através do uso direto da API. Foi desenvolvida a classe Comm que utiliza esta API para implementar a comunicação de forma mais direta e simplificada. Todos os procedimentos necessários da API serão detalhados durante a explicação da classe que os utiliza.

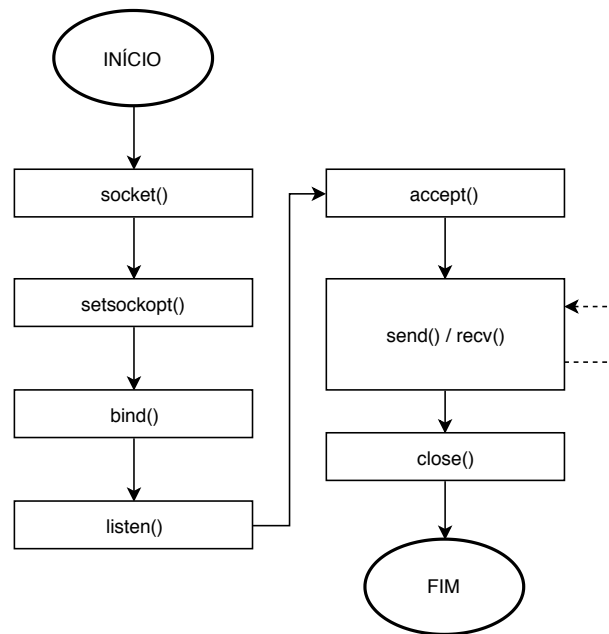


Figura 5.2 – Fluxograma dos processos para implementar o servidor no microcontrolador através da API de *socket*.

No código fonte principal a classe Comm é utilizada através dos procedimentos listados a seguir. Durante esta Seção será discutida a implementação destes processos e também será indicado onde cada procedimento é utilizado em *main.cpp*

- Instanciar objeto da classe de comunicação
- Aceitar conexão com o cliente
- Receber parâmetros
- Imprimir dados recebidos
- Receber estado da planta
- Enviar dados da saída da lei de controle
- Encerrar conexão com o cliente

5.3.1 Instanciar objeto da classe de comunicação

O primeiro procedimento é instanciar um objeto da classe de comunicação. Este ocorre *main.cpp* no seguinte fragmento de código:

```

1  int main()
2  {
3      Comm comm;
4      //(...)
5      return 0;
6  }

```

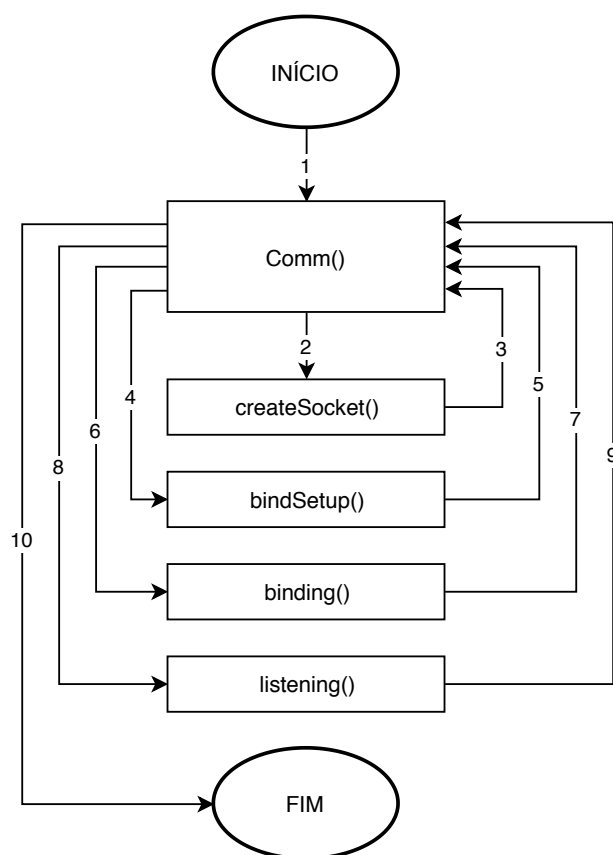


Figura 5.3 – Fluxograma dos procedimentos que ocorrem quando objeto da classe de comunicação é instanciado.

Pode-se observar que em (1) é quando o objeto é instanciado no código fonte principal, conforme o fluxograma da Fig. 5.3. O próximo procedimento ocorre em (2) quando o construtor da classe invoca a primeira função *createSocket()*. Sua implementação é mostrada a seguir:

```

1      void Comm::createSocket ()
2      {
3          // Criar o socket e verificar se houve erro neste procedimento
4          if((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
5          {
6              perror("Socket's creation failed");
7              exit(EXIT_FAILURE);
8          }
9
10         // Configura socket para reutilizar endereco e porta evitando ...
           erro de "endereço já utilizado"
11         if(setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | ...
           SO_REUSEPORT, (char* ) &opt, sizeof(opt)))
12         {
13             perror("setsockopt failed");
14             exit(EXIT_FAILURE);
15         }
16     }

```

O propósito de *createSocket()* é inicializar um *socket* no servidor que será responsável por “ouvir” solicitação de conexão de cliente. Para isso, este método da classe utiliza a função *socket()* da API de *sockets* que é o primeiro procedimento que deve-se realizar no servidor, conforme indicado no fluxograma da Fig. 5.2.

São atribuídos três argumentos para a função *socket()*³. O primeiro argumento determina qual é o domínio da comunicação. *AF_INET* é uma *flag* da API que especifica para a função que o domínio é o Protocolo de Internet versão 4 (IPv4). O segundo argumento determina o tipo da comunicação que pode ser TCP ou UDP. *SOCK_STREAM* especifica o tipo como TCP. O terceiro argumento declara um determinado protocolo para ser usado em adição com o *socket*. Segundo a implementação da API para um socket TCP deve-se utilizar o valor 0 como argumento⁴.

A função retorna um inteiro que é descritor de *socket*, análogo a um descritor de

³Disponível em: <<http://man7.org/linux/man-pages/man2/socket.2.html>>. Acesso em: 21 mar. 2019.

⁴Disponível em: <<http://man7.org/linux/man-pages/man7/ip.7.html>>. Acesso em: 21 mar. 2019.

arquivo. Em caso de erro ao criar o *socket* a função retorna -1. Logo, em caso de erro será atribuído este valor a *server_fd* e a verificação se este valor é menor do que zero indicará o erro imprimindo uma mensagem através da *perror()* e o programa será encerrado através da *system call: exit()* à qual pode ser utilizada para sinalizar se o processo no sistema operacional terminou com ou sem sucesso através das constantes *EXIT_SUCCESS* e *EXIT_FAILURE* que podem ser atribuídas como argumento para a função⁵. Neste caso, é utilizado *EXIT_FAILURE* para indicar que o programa finalizou de forma não esperada.

Após executar *socket()* com sucesso o *socket* estará criado porém ainda não está pronto para uso. Para modificar esta situação é necessário utilizar a função *bind()* da API que irá configurar este *socket*, *server_fd*, o tornando funcional.

A segunda função da API utilizada em *createSocket()* é *setsockopt()*⁶. Esta função é utilizada na API para estabelecer algumas opções⁷ referentes ao *socket* utilizado. Esta é útil pois permite a reutilização do endereço IP e porta e impede a ocorrência de erros que indicam que o endereço já está em uso. Dessa forma ganha-se mais agilidade no sistema uma vez que não será necessário aguardar o sistema operacional lidar com esta questão de liberar o endereço e porta que já estavam sendo utilizados exclusivamente para esta aplicação de comunicação referente ao *hardware-in-the-loop*.

O primeiro argumento é o descritor de *socket* criado através da função *socket()*. O segundo argumento se refere a camada onde a opção que se deseja estabelecer reside. Como a opção que se deseja alterar se encontra na camada da API de *sockets* utiliza-se *SOL_SOCKET* como argumento. O terceiro e o quarto argumento se referem a qual opção se deseja alterar, especificando seu nome, e como deseja-se alterá-la, habilitando ou desabilitando-a. No terceiro argumento são especificados as *flags* *SO_REUSEADDR* que permite a reutilização de endereços locais e *SO_REUSEPORT* que permite múltiplos *sockets* serem vinculados em um mesmo endereço. O quarto argumento é uma *flag* booleana do tipo inteiro que indica que deseja-se habilitar as opções indicadas. É atribuído o endereço da variável *opt* membra da classe, definida em *Comm.hpp* da

⁵Disponível em: <<http://man7.org/linux/man-pages/man3/exit.3.html>>. Acesso em: 21 mar. 2019.

⁶Disponível em: <<http://man7.org/linux/man-pages/man2/setsockopt.2.html>>. Acesso em: 21 mar. 2019.

⁷Disponível em: <<http://man7.org/linux/man-pages/man7/socket.7.html>>. Acesso em: 21 mar. 2019.

seguinte forma:

```

1      class Comm
2      {
3          public:
4
5              //(...)
6          private:
7
8              //(...)
9              int opt = 1;
10             //(...)
11     };

```

O quinto e último argumento é o tamanho em *bytes* alocados para a variável *opt*.

Pode-se perceber que na utilização da função *setsockopt()* realiza-se conversão de tipo como por exemplo: (*char **). Este procedimento é utilizado para atribuir o argumento com o mesmo tipo que o protótipo da função *setsockopt()* exige. Para mais detalhes, verificar a documentação da função em *The Regents of the University of California* (2017).

Também é realizada verificação sobre o retorno de *setsockopt()* a fim de detectar se houve erro em realizar as configurações das opções. Em caso de sucesso esta função retorna zero e em caso de erro, retorna -1. Como na linguagem C o valor zero é interpretado como falso e qualquer valor não nulo é considerado como verdadeiro, então os códigos dentro do segmento *if()* só serão executados caso a função retorne -1, pois este será interpretado como verdadeiro e assim será impressa mensagem indicando onde ocorreu o erro e o programa será encerrado por *exit()*. Quando *setsockopt()* for executada com sucesso será retornado 0 e este será interpretado como falso e os comandos referentes ao *if()* não serão executados.

Neste ponto a execução de *createSocket()* é finalizada. O próximo procedimento que será executado é a função *bindSetup()*. Seu propósito é configurar uma *struct* que será utilizada como argumento para a próxima função da API: *bind()*⁸. Sua implementação é dada a seguir:

⁸Disponível em: <<http://man7.org/linux/man-pages/man2/bind.2.html>>. Acesso em: 15 mar. 2019.

```

1  void Comm::bindSetup()
2  {
3      // Tipo do endereço: IPv4
4      address.sin_family = AF_INET;
5
6      /* Endereço de Internet IP: definir o endereço do host como
7      INADDR_ANY (0.0.0.0) implica que qualquer endereço poderá
8      ser vinculado. */
9      address.sin_addr.s_addr = INADDR_ANY;
10
11     // Porta na rede. PORT = 8080
12     address.sin_port = htons(PORT);
13 }

```

A *struct address* é definida como membra da classe Comm da seguinte forma:

```

1  class Comm
2  {
3      public:
4
5          //(...)
6      private:
7
8          //(...)
9          struct sockaddr_in address;
10         //(...)
11 };

```

O método *bindSetup()* configura o atributo *sin_family* da *struct address* que é do tipo *struct sockaddr_in* como *AF_INET* especificando que o endereço é do tipo IPv4. A segunda configuração é referente ao atributo *s_addr* que é membro da *struct in_addr*. Como a *struct sin_addr* é uma *struct in_addr* então *s_addr* é membra de *sin_addr*. Este parâmetro, *INADDR_ANY*, especifica que qualquer endereço poderá ser utilizado para ser vinculado ao *socket* (neste caso: *server_fd*) quando utilizar-se a função *bind()*. O endereço que será, de fato,

utilizado para o servidor é o endereço IP do microcontrolador configurado como IP estático. Neste trabalho o IP utilizado é 10.0.33.236, como é configurado que qualquer endereço é válido, então este endereço poderá ser utilizado sem problema. O último parâmetro configurado é *sin_port*. Através dele se especifica qual será a porta utilizada pelo servidor. Neste trabalho utiliza-se a porta 8080. Para estabelecer este valor deve-se utilizar a função *htons()*⁹. Esta função é responsável por ordenar os *bytes* de um dado do tipo inteiro em uma determinada ordem conhecida como “*network byte order*” que é o padrão adotado em redes¹⁰. Desta forma o valor da porta é armazenado corretamente em *sin_port* e a execução de *bindSetup()* é finalizada. Os parâmetros de endereço IP e porta devem ser conhecidos para que o cliente os utilize para solicitar conexão com servidor posteriormente.

O próximo procedimento é executar a função *binding()*. Sua implementação é a seguinte:

```

1      void Comm::binding()
2      {
3          if(bind(server_fd, (struct sockaddr *)&address, ...
4              sizeof(address)) < 0)
5          {
6              perror("bind failed");
7              exit(EXIT_FAILURE);
8          }
9      }

```

Nesta função é utilizada *bind()*, terceiro procedimento necessário indicado no fluxograma da Fig. 5.2. A finalidade desta, é vincular o *socket* (*server_fd*) ao endereço do servidor, especificado na *struct address*, e ao número da porta que este utilizará, parâmetros que foram configurados através de *bindingSetup()*.

Os argumentos atribuídos para *bind()* são: o descritor de *socket*, *server_fd*, criado anteriormente, a *struct address*, e o tamanho, em *bytes*, desta *struct*. O retorno desta função

⁹Disponível em: <<https://linux.die.net/man/3/htons>>. Acesso em: 21 jul. 2018.

¹⁰Disponível em: <https://www.ibm.com/support/knowledgecenter/en/SSB27U_6.4.0/com.ibm.zvm.v640.kiml0/asonetw.htm>. Acesso em: 21 jul. 2018.

é utilizado para checar se houve erro em sua execução. É retornado 0 em caso de sucesso e -1 quando houver erro e neste caso é impressa mensagem indicando erro e o programa é encerrado.

Pode-se perceber que na utilização da função *bind()* é realizada conversão de tipo como por exemplo: *(struct sockaddr *)*. Este procedimento é utilizado para atribuir o argumento com o mesmo tipo que o protótipo da função *bind()* exige. Para mais detalhes, verificar a documentação da função em *Faith (2019)*.

Neste ponto, o servidor já está configurado, estabelecido seu endereço e sua porta. O *socket server_fd* criado e configurado está apto para ser utilizado para “escutar” solicitação de conexão de um cliente. Para efetivamente colocar este *socket* do servidor para “ouvir” solicitações coloca-se o servidor em um estado passivo onde ele aguarda por novas solicitações de conexões através do uso de *listening()*, implementada da seguinte forma:

```
1      void Comm::listening()
2      {
3          if(listen(server_fd, 1) < 0)
4          {
5              perror("listen failed");
6              exit(EXIT_FAILURE);
7          }
8      }
```

São atribuídos como argumentos o descritor de *socket* criado anteriormente e o tamanho máximo da fila de conexões pendentes de clientes para conectarem com o descritor *server_fd*. Neste caso o tamanho máximo é definido como 1 e caso existam mais do que uma conexão pendente estas serão descartadas e estes clientes poderão receber um erro indicando que a conexão com o servidor foi recusada. Como neste trabalho só haverá um cliente conectado ao servidor, então não haverá este tipo de problema.

O parâmetro retornado por *listen()*¹¹ é zero quando a função é executada sem erro e -1 quando ocorrer falha e neste caso será impressa mensagem na tela indicando onde houve erro e o programa será encerrado. Após *listening()* terminar de executar o fluxo de execução retorna

¹¹Disponível em: <<http://man7.org/linux/man-pages/man2/listen.2.html>>. Acesso em: 20 jul. 2018.

para o construtor da classe e neste momento todas as funções do construtor foram executadas e então *Comm()* será finalizada.

5.3.2 Aceitar conexão com o cliente

No código fonte principal, após a construção do objeto, o próximo momento em que a classe é utilizada é para aceitar a solicitação de conexão do cliente. Este pode ser visualizado no fragmento a seguir:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              comm.accepting();
7              //(...)
8          }
9          //(...)
10     }
```

O método *accepting()* utiliza a função *accept()*¹² que deve ser empregada para implementar o servidor de acordo com a API, conforme indicado na Fig. 5.2. A implementação de *accepting()* é a seguinte:

```

1      void Comm::accepting()
2      {
3          if((new_socket = accept(server_fd, (struct sockaddr *)&address, ...
4              (socklen_t *)&addrlen)) < 0)
5          {
6              perror("accept failed");
7              exit(EXIT_FAILURE);
8          }
9      }
```

¹²Disponível em: <<http://man7.org/linux/man-pages/man2/accept.2.html>>. Acesso em: 15 mar. 2019.

```

7         }
8         std::cout << "Connection stablished." << std::endl;
9     }

```

São atribuídos como argumentos o descritor de *socket*, *server_fd* que é aquele estabelecido em estado de escuta anteriormente. O segundo argumento é a *struct* que configura os dados de endereço do servidor. O terceiro argumento é o tamanho, em *bytes*, da *struct address*. Este tamanho é obtido na implementação da classe em *Comm.hpp* da seguinte forma:

```

1     class Comm
2     {
3     public:
4
5         //(...)
6     private:
7
8         //(...)
9         struct sockaddr_in address;
10        int addrlen = sizeof(address);
11        //(...)
12    };

```

Pode-se perceber que na utilização da função *accept()* são realizadas conversões de tipos como por exemplo: *(struct sockaddr*)* e *(socklen_t*)*. Estes procedimentos são utilizados para atribuir os argumentos com o mesmo tipo que o protótipo da função *accept()* exige. Para mais detalhes, verificar a documentação da função em *The Regents of the University of California* (2019).

A função *accept()* cria e retorna um novo *socket*, *new_socket*. Este será utilizado para realizar a comunicação entre o servidor e o cliente. O primeiro *socket*, *server_fd*, criado por *socket()* se mantêm com o propósito de ouvir conexões de possíveis novos clientes, no endereço 10.0.33.236 e porta 8080. Já o *new_socket* utilizará outra porta para se comunicar com o cliente implementado no MATLAB®

O retorno da função é utilizado para checar se houve erro em sua execução. Em caso de erro de execução a função retorna -1 e neste caso será impressa mensagem indicando erro e o programa será encerrado. Em caso de sucesso, a partir deste ponto, servidor e cliente poderão trocar dados entre si através das funções *send()*¹³ e *recv()*¹⁴.

5.3.3 Receber parâmetros

Na execução do código fonte principal, o próximo procedimento é receber os seguintes parâmetros, do cliente MATLAB®: matrizes de ganho $K1$ e Kx , coeficientes do polinômio de referência, tempo de amostragem, trimagem e número de iterações de simulação. Estes dados são recebidos em *main.cpp* através do método *receberDados()* da classe Comm da seguinte forma:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              // Receber parametros iniciais do MATLAB
8              comm.receberDados(K1Vet, sizeof(K1Vet));
9              comm.receberDados(KxVet, sizeof(KxVet));
10             comm.receberDados(CoefRefVet, sizeof(CoefRefVet));
11             comm.receberDados(&dtFromMatlab, sizeof(dtFromMatlab));
12             comm.receberDados(trimVet, sizeof(trimVet));
13             comm.receberDados(&numeroIteracoes, sizeof(numeroIteracoes));
14             //(...)
15         }
16         //(...)
17     }

```

Os argumentos de *receberDados()* são o endereço de memória da variável que armazenará o dado enviado pelo cliente e o tamanho, em *bytes*, do dado que será enviado. Neste caso

¹³Disponível em: <<https://linux.die.net/man/3/send>>. Acesso em: 21 jul. 2018.

¹⁴Disponível em: <<http://man7.org/linux/man-pages/man2/recv.2.html>>. Acesso em: 08 ago. 2018.

utiliza-se a própria quantidade de *bytes* alocada para a variável que vai armazenar o dado que será enviado pelo cliente pois esta deve ter quantidade suficiente para armazená-lo, caso contrário haverá perda de informações. A implementação de *receberDados()* na classe é mostrada a seguir:

```
1     void Comm::receberDados(void* Guardar, size_t lenGuardar)
2     {
3         recv(new_socket, Guardar, lenGuardar, 0);
4     }
```

Neste método é utilizada a função *recv()* da API. É através de seu uso que um dado é recebido do cliente no servidor. São atribuídos como argumentos para a função o descritor do *socket*, *new_socket*, criado, pela função *accept()*, responsável por estabelecer a comunicação entre os elementos; o endereço de memória da variável que armazenará o dado; o seu tamanho, em *bytes*; e, como último argumento, é atribuindo para a *flag* da função o valor zero. Por fim, considere-se que a função *recv()* se assemelha a função *read()* quando se atribui o valor nulo à esta flag. Para maiores informações, consultar a documentação de *recv()* em *The Regents of the University of California* (2017).

Desta forma, os parâmetros iniciais são recebidos no microcontrolador através do uso do método *receberDados()*.

5.3.4 Imprimir dados recebidos

O próximo momento em que a classe de comunicação é utilizada em *main.cpp* é logo após receber os parâmetros iniciais do MATLAB®. A classe é utilizada para imprimir na tela os dados recebidos a fim de que o usuário possa verificar se os dados recebidos conferem com os dados enviados. O uso acontece da seguinte forma:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              // Imprimir parametros iniciais do MATLAB
8              comm.printData("Vetor K1 from matlab", K1Vet, iK1*jK1);
9              comm.printData("Vetor Kx from matlab", KxVet, iKx*jKx);
10             comm.printData("Vetor de Coeficientes de referencia from ...
                matlab", CoefRefVet, icoefRef*jcoefRef);
11             comm.printData("dt from matlab", &dtFromMatlab, 1);
12             comm.printData("Vetor trim from matlab", trimVet, iTrim*jTrim);
13             comm.printData("numeroIteracoes from matlab", ...
                &numeroIteracoes, 1);
14             //(...)
15         }
16         //(...)
17     }

```

Todos os dados são tratados como vetores, com exceção dos escalares *dtFromMatlab* e *numeroIteracoes* que são considerados vetores de uma posição, e assim deve ser informado quantas posições estes possuem.

A implementação da função é mostrada a seguir:

```

1      void Comm::printData(const char *label, double *vetor, size_t sizeVet)
2      {
3          std::cout << label << ": " << std::endl;
4          for(int i = 0; i < sizeVet; i++)
5          {
6              std::cout << i << ": " << *(vetor++) << std::endl;
7          }
8          std::cout << "-----" << std::endl;
9      }

```

Quando a função é invocada, ela recebe a *string label* e a imprime na tela. Seu próximo procedimento é imprimir o índice do vetor e o valor respectivo, este procedimento é feito para cada índice do vetor. Por fim, é impressa uma mensagem delimitando o fim dos dados impressos.

5.3.5 Receber estado da planta

Dando sequência na execução do código fonte principal, a próxima ocorrência da utilização da classe é quando recebe-se no servidor o estado da planta enviado pelo MATLAB® na execução de cada iteração do *hardware-in-the-loop*. Este momento ocorre da seguinte forma:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              for(int i = 0; i < numeroIteracoes; i++)
8              {
9                  // Ler XX
10                 comm.receberDados(XXVet, sizeof(XXVet));
11                 //(...)
12             }
13             //(...)
14         }
15         //(...)
16     }

```

Neste caso, são atribuídos o endereço da variável que armazenará o estado e o tamanho, em *bytes*, desta.

5.3.6 Enviar dados da saída da lei de controle

O próximo uso da classe de comunicação ocorre após as saídas da lei de controle serem calculadas, convertidas para *duty cycles* equivalentes e estabelecidos estes dados nos motores.

É o último procedimento executado em uma iteração do *hardware-in-the-loop*. Este processo ocorre da seguinte forma:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              for(int i = 0; i < numeroIteracoes; i++)
8              {
9                  //(...)
10                 comm.enviarDados(ptrOutCntToMatlab, ...
11                                lqr.getOutputCtlSize());
12             }
13             //(...)
14         }
15     }

```

O método *enviarDados()* é utilizado para enviar para o cliente, MATLAB®, as saídas, velocidades em rad/s, calculadas pela lei de controle. Sua forma de utilização é semelhante a *receberDados()*. Neste caso o primeiro argumento atribuído ao método é o endereço da variável que deseja-se enviar e o segundo é o tamanho, em *bytes*, desta. Este tamanho é obtido através do método *getOutputCtlSize()* da classe LQR que será detalhado na Seção sobre esta classe.

A implementação de *enviarDados()* é a seguinte:

```

1      void Comm::enviarDados(void* Enviar, size_t lenEnviar)
2      {
3          send(new_socket, Enviar, lenEnviar, 0);
4      }

```

Este método utiliza a função *send()* da API de *socket* indicado na Fig. 5.2. É através


```

1     void Comm::closeSocket ()
2     {
3         int ec;
4         ec = close(new_socket);
5         if(ec == 0)
6             std::cout << "Closing connection with the client." << ...
7             std::endl;
8         else
9             std::cout << "error closing the socket." << std::endl;
10    }

```

O método utiliza a função `close()`¹⁵ da API, indicada na Fig. 5.2. Como argumento, lhe é atribuído o descritor de *socket* que deseja-se fechar, encerrando, desta maneira, a conexão com um cliente. A função retorna -1 em caso de erro em sua execução ou 0 em caso de sucesso. Este retorno é utilizado para verificar erro no procedimento de fechar o *socket*. Quando é retornado 0 para variável `ec` é exibida mensagem indicando que a conexão foi encerrada com o cliente. Em caso de erro é exibida mensagem indicando que não houve sucesso em realizar este procedimento.

Neste ponto todos os procedimentos de comunicação com um cliente foram realizados. Caso o usuário deseje se conectar novamente, enviando novos parâmetros pelo MATLAB®, por exemplo, a maior parte dos procedimentos descritos nesta Seção se repetirão. O único passo que não será realizado é configurar o servidor e o colocar em estado de escuta por conexões.

5.4 Classe LQR

Nesta Seção será detalhada a classe LQR implementada nos arquivos *lqr.hpp* e *lqr.cpp*. Essa classe é responsável por implementar a lei de controle e calcular o vetor de entrada da planta.

No código fonte principal a classe LQR é utilizada nos procedimentos listados a seguir. Durante esta Seção será discutida a implementação destes processos e indicado onde cada procedimento é utilizado em *main.cpp*

- Instanciar objeto da classe da lei de controle

¹⁵Disponível em: <<https://linux.die.net/man/3/close>>. Acesso em: 21 jul. 2018.

- Definir o estado da planta no controlador
- Computar as saídas da lei de controle
- Obter as saídas da lei de controle
- Retornar dimensão, em *bytes*, do vetor das saídas do controlador

5.4.1 Instanciar objeto da classe *LQR*

O primeiro momento em que a classe é utilizada é no processo de instanciar objeto desta no código fonte principal (*main.cpp*). São atribuídos como argumentos para o construtor da classe os parâmetros de controle: *K1Vet*, *KxVet*, *CoefRefVet*, *dtFromMatlab* e *trimVet*. Com exceção de *dtFromMatlab*, que é um escalar, todas as demais variáveis são vetores.

O fluxograma da Fig. 5.4 mostra os processos que ocorrem internamente na classe no momento que ocorre a instancialização de um objeto no código fonte principal (*main.cpp*). O objetivo que se deseja obter através destes procedimentos é, de forma resumida, inicializar dentro de um objeto da classe os parâmetros de controle. Para isso os dados são recebidos na forma de vetor e depois reconstruídos na forma de matriz. Após a reconstrução, as matrizes são convertidas para um formato matricial implementado na biblioteca *open source Armadillo*¹⁶, utilizada para álgebra linear e computação científica. Seu uso proporciona maior facilidade, uma vez que seu uso se assemelha a sintaxe e forma de trabalho com matrizes do MATLAB®, e rapidez em cálculos matriciais.

O último procedimento é computar o sinal de entrada do controlador com base nos coeficientes do polinômio que compõem o sinal de referência. Neste trabalho, essa referência é constituída por polinômios de primeiro grau para as três coordenadas x , y e z que podem variar ou não em função do tempo.

¹⁶Disponível em: <<http://arma.sourceforge.net/>>. Acesso em: 21 ago. 2018.

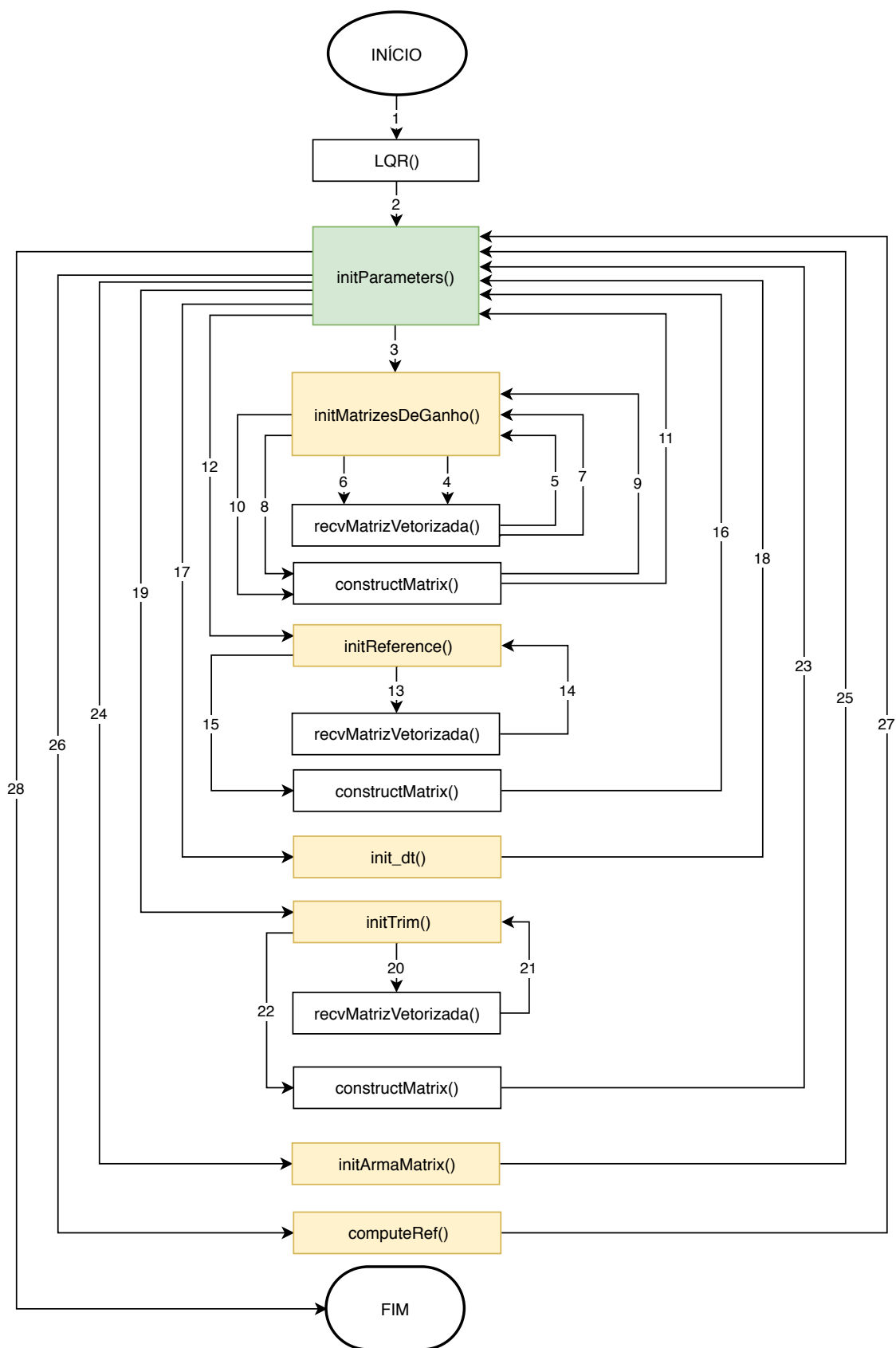


Figura 5.4 – Fluxograma dos procedimentos que ocorrem internamente no momento de instanciar um objeto da classe LQR.

O objeto da classe LQR é instanciado no código fonte principal onde são atribuídos os parâmetros de controle ao mesmo, processo que ocorre em (1). O fragmento de código abaixo é retirado de *main.cpp* onde ocorre o processo descrito:

```
1      int main()
2      {
3          // (...)
4          while(true)
5          {
6              // (...)
7              LQR lqr(K1Vet, KxVet, CoefRefVet, dtFromMatlab, trimVet);
8              // (...)
9          }
10         // (...)
11         return 0;
12     }
```

Dentro da classe o construtor invoca a função *initParameters()*, passo indicado em (2) na Fig. 5.4. O propósito desta função é inicializar dentro da classe os parâmetros de controle atribuídos na criação do objeto. Os argumentos desta função são os endereços dos vetores *K1VetMatlab*, *KxVetMatlab*, *CoefRefVetMatlab* e *trimVetMatlab* e o valor do escalar *dtMatlab*.

```

1   LQR::LQR(double *K1VetMatlab, double *KxVetMatlab, double ...
      *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
2   {
3       initParameters(K1VetMatlab, KxVetMatlab, CoefRefVetMatlab, ...
          dtMatlab, trimVetMatlab);
4   }

```

Em (3) a função *initParameters()* invoca a *initMatrizesDeGanho()* atribuindo como argumentos os endereços dos de *K1VetMatlab* e *KxVetMatlab*.

```

1   void LQR::initParameters(double *K1VetMatlab, double *KxVetMatlab, ...
      double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
2   {
3       // K1 e Kx
4       initMatrizesDeGanho(K1VetMatlab, KxVetMatlab);
5       // (...)
6   }

```

O método *initMatrizesDeGanho()* tem papel de gerenciar as funções que irão montar as matrizes de ganho. Esta invoca *recvMatrizVetorizada()* em (4) onde são enviados o endereço do vetor onde está armazenado o dado que se deseja montar dentro da classe, a dimensão do vetor e uma *string* para sinalizar dentro da função qual dado será montado.

```

1      void LQR::initMatrizesDeGanho(double *K1VetMatlab, double *KxVetMatlab)
2      {
3          recvMatrizVetorizada(K1VetMatlab, iK1*jK1, "K1");
4          //(...)
5      }

```

A função *recvMatrizVetorizada()* é utilizada para transportar um dado enviado a ela e armazenar o mesmo internamente dentro da classe LQR. Neste caso, é enviado pelo MATLAB® o vetor representando a matriz de ganho *K1*, vale destacar que esses ganhos são calculados a partir das equações de Ricatti (2.17). Este dado é recebido no microcontrolador pela classe de comunicação Comm que armazena no vetor, declarado no código fonte principal *main.cpp*, *K1Vet*. O endereço deste vetor posteriormente é atribuído como um dos argumentos para o construtor da classe LQR. Este por sua vez irá passar o endereço deste vetor para a função *initParameters()* e esta repassará o endereço para *initMatrizesDeGanho()* que repassará o endereço para *recvMatrizVetorizada()*. Neste momento, *recvMatrizVetorizada()* irá acessar o endereço de memória onde está contido o vetor e armazenar os dados deste em um vetor membro da classe LQR (*K1Vetorizado*).

Todo esse procedimento é realizado para separar as atribuições de cada classe. Os endereços dos vetores são repassados dentre essas funções para evitar que sejam criadas cópias diversas vezes dos vetores aumentando o custo computacional do sistema.

Observação: as declarações das *flags* e demais variáveis mostradas no fragmento de código seguinte serão representadas como um bloco de comentários nas próximas aparições do método *recvMatrizVetorizada()* ao longo desta Seção.

```

1      void LQR::recvMatrizVetorizada(double *vetFromMatlab, size_t ...
2          size_vetFromMatlab, const char *paramControle)
3      {
4          // Atribui o vetor recebido enviado pelo MATLAB para um vetor ...
5          // temporario dentro da classe
6
7          // Declaracao de flags

```

```

6      bool compareStringK1, compareStringKx, compareStringRef,
7      compareStringCoefRef, compareStringSV, compareStringTrim;
8
9      // Atribuicao de estado para o funcionamento correto da logica ...
      utilizada
10     compareStringK1 = true;
11     compareStringKx = true;
12     compareStringRef = true;
13     compareStringCoefRef = true;
14     compareStringSV = true;
15     compareStringTrim = true;
16
17     // Analise para determinar qual parametro sera montado
18     compareStringK1 = strcmp(paramControle, "K1");
19     compareStringKx = strcmp(paramControle, "Kx");
20     compareStringRef = strcmp(paramControle, "Ref");
21     compareStringCoefRef = strcmp(paramControle, "CoefRef");
22     compareStringSV = strcmp(paramControle, "SV");
23     compareStringTrim = strcmp(paramControle, "trim");
24
25     if(!compareStringK1)
26     {
27         for(size_t i = 0; i < size_vetFromMatlab; i++)
28         {
29             K1Vetorizado[i] = vetFromMatlab[i];
30         }
31     }
32     //(...)
33 }

```

A função *recvMatrizVetorizada()* é construída de forma que seja capaz de utilizá-la para armazenar diversos vetores como os relacionados as matrizes de ganho $K1$ e Kx , trimagem e outros utilizando-se as *flags* de prefixo “*compareString*” que especifica diferentes vetores de diversos tamanhos. No fragmento acima os dados contidos em *K1Vet* serão armazenados no vetor *K1Vetorizado*. Quando o armazenamento terminar, a função *recvMatrizVetorizada()* termina

base um vetor. É neste momento que ocorre o inverso da adequação de uma matriz em um vetor, procedimento exemplificado na Fig. 5.5. Neste caso será reconstruída a matriz $K1$ com as mesmas dimensões da matriz que foi definida e enviada pelo MATLAB® (4x3).

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \end{bmatrix} \Rightarrow \begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}$$

Figura 5.5 – Ilustração do processo de construção de matriz com base em um vetor.

São atribuídos como argumentos da função quantas linhas, colunas e uma *string* que especifica qual será a matriz montada, desta forma, detalhando para função, qual vetor deverá ser utilizado. $iK1$ e $jK1$ são diretivas definidas em *lqr.hpp* que definem o número de linhas e colunas da matriz $K1$.

```

1      void LQR::initMatrizesDeGanho(double *K1VetMatlab, double *KxVetMatlab)
2      {
3          recvMatrizVetorizada(K1VetMatlab, iK1*jK1, "K1");
4          recvMatrizVetorizada(KxVetMatlab, iKx*jKx, "Kx");
5
6          constructMatrix(iK1, jK1, "K1");
7          // (...)
8      }
```

A definição de *constructMatrix()* é exibida a seguir. No fragmento mostado, será montada a matriz $K1$, membra da classe LQR, baseada no vetor $K1Vetorizado$.

Neste método são utilizadas *flags* de prefixo “*compareString*” para especificar qual matriz será construída, pois estas possuem dimensões distintas.

Observação: as declarações das *flags* e demais variáveis mostradas no fragmento de código seguinte serão representadas como um bloco de comentários nas próximas aparições do método *constructMatrix()* ao longo desta Seção.

```

1     void LQR::constructMatrix(size_t linSize, size_t colSize, const ...
      char *matrix)
2     {
3         // Argumento 'char *matrix' especifica a matriz a ser montada
4         // Variavel auxiliar
5         int aux = 0;
6
7         // Declaracao de flags
8         bool compareStringK1, compareStringKx, compareStringRef,
9         compareStringCoefRef, compareStringSV, compareStringTrim;
10
11        // Atribuicao de estado para o funcionamento correto da logica ...
          utilizada
12        compareStringK1 = true;
13        compareStringKx = true;
14        compareStringRef = true;
15        compareStringCoefRef = true;
16        compareStringSV = true;
17        compareStringTrim = true;
18
19        // Analise para determinar qual parametro sera montado
20        compareStringK1 = strcmp(matrix, "K1");
21        compareStringKx = strcmp(matrix, "Kx");
22        compareStringRef = strcmp(matrix, "Ref");
23        compareStringCoefRef = strcmp(matrix, "CoefRef");
24        compareStringSV = strcmp(matrix, "SV");
25        compareStringTrim = strcmp(matrix, "trim");
26
27        if(!(compareStringK1))
28        {
29            for(size_t j = 0; j < colSize; j++)
30            {
31                for(size_t i = 0; i < linSize; i++)
32                {
33                    K1[i][j] = K1Vetorizado[aux];

```

O processo que ocorre em (9) acontece desta forma:

```
1 void LQR::initMatrizesDeGanho(double *KlVetMatlab, double *KxVetMatlab)
2 {
3     recvMatrizVetorizada(KlVetMatlab, iKl*jKl, "Kl");
4     recvMatrizVetorizada(KxVetMatlab, iKx*jKx, "Kx");
5
6     constructMatrix(iKl, jKl, "Kl");
7     constructMatrix(iKx, jKx, "Kx");
8 }
```

```

1 void LQR::constructMatrix(size_t linSize, size_t colSize, const ...
    char *matrix)
2 {
3     /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
4      * Declaracoes de flags e demais variaveis do metodo *
5      * * * * * * * * * * * * * * * * * * * * * * * * * */
6     //(...)
7     if(!compareStringKx())
8     {
9         for(size_t j = 0; j < colSize; j++)
10            {

```

```

11         for(size_t i = 0; i < linSize; i++)
12         {
13             Kx[i][j] = KxVetorizado[aux];
14             aux++;
15         }
16     }
17 }
18 // (...)
19 }

```

Terminado o procedimento (10) volta-se ao ponto inicial, *initParameters()*, em (11). Neste momento as matrizes de ganho estão internalizadas dentro do objeto. O próximo passo é construir a matriz de referência através dos processos (12), (13), (14) e (15).

Da mesma forma que a função *initMatrizesDeGanho()*, *initReference()* tem papel de gerenciar os procedimentos para instanciar o sinal de referência. Em (12) esta função é invocada da seguinte forma:

```

1  void LQR::initParameters(double *KlVetMatlab, double *KxVetMatlab, ...
    double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
2  {
3      // Kl e Kx
4      initMatrizesDeGanho(KlVetMatlab, KxVetMatlab);
5      // Referencia
6      initReference(CoefRefVetMatlab);
7      // (...)
8  }

```

O argumento *CoefRefVetMatlab* atribuído a função *initReference()* armazena o endereço do vetor dos coeficientes de referência (*CoefRefVet*). No instante que o objeto da classe LQR é instanciado este endereço é armazenado no parâmetro *CoefRefVetMatlab*. O método *initReference()* recebe como argumento de entrada um endereço de vetor.

O procedimento (13) ocorre no fragmento seguinte:


```

1      void LQR::constructMatrix(size_t linSize, size_t colSize, const ...
        char *matrix)
2      {
3          /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
4             * Declaracoes de flags e demais variaveis do metodo *
5             * * * * * * * * * * * * * * * * * * * * * * * * * * * */
6          //(...)
7          if(!(compareStringCoefRef))
8          {
9              for(size_t j = 0; j < colSize; j++)
10             {
11                 for(size_t i = 0; i < linSize; i++)
12                 {
13                     coefRef[i][j] = coefRefVetorizado[aux];
14                     aux++;
15                 }
16             }
17         }
18         //(...)
19     }

```

```
1 void LQR::initParameters(double *KlVetMatlab, double *KxVetMatlab, ...
    double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
```

```

2      {
3          // K1 e Kx
4          initMatrizesDeGanho(K1VetMatlab, KxVetMatlab);
5          // Referencia
6          initReference(CoefRefVetMatlab);
7          // Tempo dt
8          init_dt(dtMatlab);
9          //(...)
10     }

```

Após a invocação da função, o dado é armazenado internamente da seguinte forma:

```

1      void LQR::init_dt(double dtMatlab)
2      {
3          dt = dtMatlab;
4      }

```

Quando termina a execução desta função volta-se ao ponto inicial em (18) onde será invocada a função *initTrim()* responsável por inicializar os dados referentes a trimagem em (19). É atribuído a função (*initTrim()*) o endereço de memória do vetor *trimVet* do código fonte principal.

```

1      void LQR::initParameters(double *K1VetMatlab, double *KxVetMatlab, ...
2                                double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
3      {
4          // K1 e Kx
5          initMatrizesDeGanho(K1VetMatlab, KxVetMatlab);
6          // Referencia
7          initReference(CoefRefVetMatlab);
8          // Tempo dt
9          init_dt(dtMatlab);
10         // Trim
11         initTrim(trimVetMatlab);
12         //(...)

```

A invocação de *recvMatrizVetorizada()*:

O processo que o ocorre em (20) procede da seguinte forma:

Após este procedimento, retorna-se para *initTrim()* onde é invocada *constructMatrix()*:


```

1  void LQR::initParameters(double *K1VetMatlab, double *KxVetMatlab, ...
    double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
2  {
3      // K1 e Kx
4      initMatrizesDeGanho(K1VetMatlab, KxVetMatlab);
5      // Referencia
6      initReference(CoefRefVetMatlab);
7      // Tempo dt
8      init_dt(dtMatlab);
9      // Trim
10     initTrim(trimVetMatlab);
11     // Inicilizar as matrizes convertidas para Armadillo
12     initArmaMatrix();
13     //(...)
14 }

```

Os procedimentos que ocorrem em (24) convertem as matrizes $K1$, Kx , $trim$ e $coefRef$ para o formato *Armadillo* para que sejam utilizadas para realizar os cálculos da lei de controle.

```

1  void LQR::initArmaMatrix()
2  {
3      // Define o tamanho de todas matrizes e as inicializa com zero
4      armaK1.zeros(iK1, jK1);
5      armaKx.zeros(iKx, jKx);
6      armaRef.zeros(iRef, jRef);
7      armaSV.zeros(iSV, jSV);
8      armaTrim.zeros(iTrim, jTrim);
9      armaError.zeros(iError, jError);
10     armaOutputControl.zeros(iOutputControl, jOutputControl);
11     armaCoefRef.zeros(icoefRef, jcoefRef);
12
13     // Atribuir valores para K1 no escopo do Armadillo
14     for(size_t j = 0; j < jK1; j++)
15     {

```

```

16         for(size_t i = 0; i < iK1; i++)
17         {
18             armaK1(i, j) = K1[i][j];
19         }
20     }
21
22     // Atribuir valores para Kx no escopo do Armadillo
23     for(size_t j = 0; j < jKx; j++)
24     {
25         for(size_t i = 0; i < iKx; i++)
26         {
27             armaKx(i, j) = Kx[i][j];
28         }
29     }
30
31     // Atribuir valores para trim no escopo do Armadillo
32     for(size_t j = 0; j < jTrim; j++)
33     {
34         for(size_t i = 0; i < iTrim; i++)
35         {
36             armaTrim(i, j) = trim[i][j];
37         }
38     }
39
40     // Atribuir valores para COEFREF no escopo do Armadillo
41     for(size_t j = 0; j < jcoefRef; j++)
42     {
43         for(size_t i = 0; i < icoefRef; i++)
44         {
45             armaCoefRef(i, j) = coefRef[i][j];
46         }
47     }
48 }

```

Em (25) retorna-se ao ponto inicial, após o fim da execução de *initArmaMatrix()*, onde será invocada a última função de todo o processo, em (26): *computeRef()* responsável por

calcular, utilizando os coeficientes dos polinômios, o sinal de referência.

```

1      void LQR::initParameters(double *KlVetMatlab, double *KxVetMatlab, ...
        double *CoefRefVetMatlab, double dtMatlab, double *trimVetMatlab)
2      {
3          // Kl e Kx
4          initMatrizesDeGanho(KlVetMatlab, KxVetMatlab);
5          // Referencia
6          initReference(CoefRefVetMatlab);
7          // Tempo dt
8          init_dt(dtMatlab);
9          // Trim
10         initTrim(trimVetMatlab);
11         // Inicializar as matrizes convertidas para Armadillo
12         initArmaMatrix();
13         // Computar a referencia
14         computeRef();
15     }

```

Após a invocação em (28) o cálculo da referência ocorrerá e será armazenado na matriz *armaRef*. O primeiro procedimento de *computeRef()* é calcular o instante de tempo, *t* que é uma variável interna da classe de controle, para que este seja usado nos casos onde a referência varia no tempo. Em seguida para cada eixo, obtêm-se os coeficientes armazenados na matriz *coefRef*, depois calcula-se o valor da referência para cada eixo e, por fim, atualiza-se o tempo inicial para que no próximo uso de *computeRef()* o tempo seja calculado levando em conta o tempo transcorrido.

```

1      void LQR::computeRef()
2      {
3          // Definir o instante de tempo atual
4          t = t0 + dt;
5
6          // x: Ax*t + Bx

```

```

7      double Ax = coefRef[0][0];
8      double Bx = coefRef[0][1];
9      armaRef(0, 0) = Ax*t + Bx;
10
11     // y: Ay*t + By
12     double Ay = coefRef[1][0];
13     double By = coefRef[1][1];
14     armaRef(1, 0) = Ay*t + By;
15
16     // z: Az*t + Bz
17     double Az = coefRef[2][0];
18     double Bz = coefRef[2][1];
19     armaRef(2, 0) = Az*t + Bz;
20
21     // Atualizacao de t - pegar o ultimo valor para a proxima iteracao
22     t0 = t;
23 }

```

Em (27) retorna-se pela última vez em *initParameters()* e em (28) chega-se ao fim dos procedimentos que ocorrem no momento de instanciar um objeto da classe LQR em *main.cpp*.

5.4.2 Definir o estado da planta no controlador

Nesta Seção será detalhado o procedimento de envio do estado da planta para o controlador que ocorre em cada iteração da simulação HIL. Os procedimentos poderão ser melhor compreendidos pelo fluxograma da Fig. 5.6.

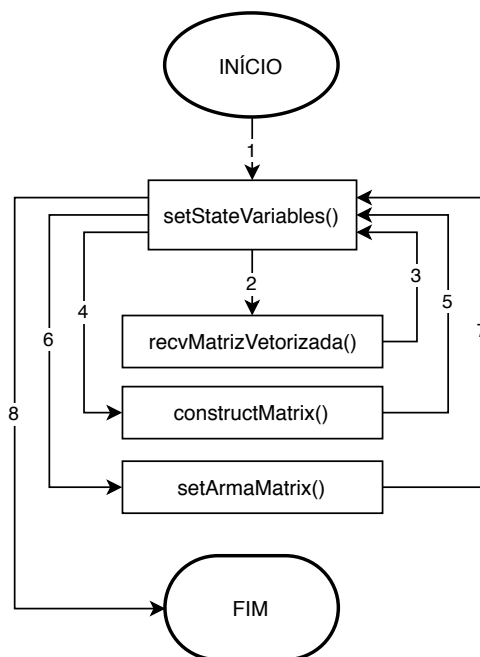


Figura 5.6 – Fluxograma dos procedimentos realizados para estabelecer o estado da planta no controlador.

O estado da planta varia constantemente durante o uso do sistema e estas variações devem ser enviadas ao controlador para que o mesmo seja capaz de calcular as saídas adequadas para se obter uma atitude da aeronave desejada. Então, para realizar este envio constante, utiliza-se o método *setStateVariables()* membro da classe LQR. O envio ocorre dentro do laço descrito em “Número de iterações” referente ao código principal *main.cpp*. No fragmento abaixo pode-se ver o seu uso em *main.cpp* que está representado na Fig. 5.6 em (1).

```

1  int main()
2  {
3      //(...)
4      while(true)
5      {
6          //(...)
7          for(int i = 0; i < numeroIteracoes; i++)
8          {
9              //(...)
10             // Calcular controle
11             lqr.setStateVariables(XXVet, iSV*jSV);

```



```

10         {
11             for(size_t i = 0; i < linSize; i++)
12             {
13                 SV[i][j] = SVVetorizado[aux];
14                 aux++;
15             }
16         }
17     }
18     //(...)
19 }

```

Finalizada sua execução retorna-se para *setStateVariables()* em (5), esta por sua vez invoca *setArmaMatrix()* em (6), para converter a matriz de estado, atribuindo-lhe como argumento a referência da matriz *armaSV*, no padrão *Armadillo* e o número de linhas e colunas desta. Deve-se destacar que ao instanciar o objeto da classe a função *initArmaMatrix()* criou uma matriz em *Armadillo* (*armaSV*) de dimensão 12x1 e a inicializou com zeros. Será ela que armazenará os dados de estado e é por isso que é passado a referência dela na função *setArmaMatrix()*

```

1  void LQR::setStateVariables(double *XXvetFromMatlab, size_t ...
    sizeColXXvetFromMatlab)
2  {
3      recvMatrizVetorizada(XXvetFromMatlab, sizeColXXvetFromMatlab, ...
        "SV");
4
5      constructMatrix(iSV, jSV, "SV");
6
7      setArmaMatrix(armaSV, iSV, jSV);
8  }

```

A função armazena os dados em *armaSV* com base na matriz *SV* da seguinte forma:

```

1  void LQR::setArmaMatrix(arma::mat &ArmaMatrix, size_t sizeLin, ...
    size_t sizeCol)

```

```

2      {
3          // Define iterativamente novos valores de SV (State Variables)
4          for(size_t j = 0; j < sizeCol; j++)
5          {
6              for(size_t i = 0; i < sizeLin; i++)
7              {
8                  ArmaMatrix(i, j) = SV[i][j];
9              }
10         }
11     }

```

Finalizada sua execução retornar-se em *setStateVariables()* em (7) terminando assim os procedimentos referentes a definir o estado da planta no controlador em (8).

5.4.3 Computar as saídas da lei de controle

A partir deste ponto, a classe possui todos os dados necessários para calcular a ação de controle utilizando um LQR.

Os processos que ocorrem no cálculo são exibidos no fluxograma da Fig. 5.7.

A invocação da função *computeControl()* é realizada no código fonte principal dentro do laço descrito em “Número de iterações”. No fragmento a seguir pode-se ver o seu uso em *main.cpp* representado na Fig. 5.7 em (1).

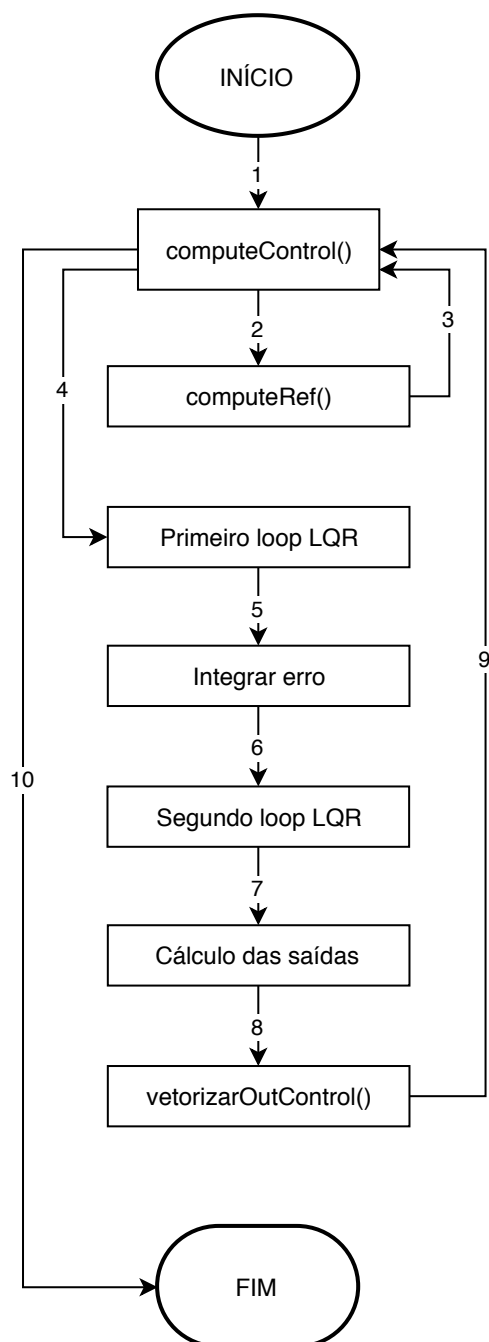


Figura 5.7 – Fluxograma do processo de cálculo das saídas realizado pelo controlador.

```

1  int main()
2  {
3      // (...)
4      while(true)
5      {
6          // (...)

```

```

7         for(int i = 0; i < numeroIteracoes; i++)
8         {
9             //(...)
10            // Calcular controle
11            //(...)
12            lqr.computeControl();
13            //(...)
14        }
15        //(...)
16    }
17    return 0;
18 }

```

O próximo procedimento é calcular o sinal de referência através da função *computeRef()* em (2) que após finalizar sua execução retorna para *computeControl()* em (3). Os procedimentos em (4), (5), (6) e (7) ocorrem dentro de *computeControl()*. Estes procedimentos utilizam os recursos da biblioteca *Armadillo* para realizar multiplicações matriciais de forma direta e para criar uma sub-matriz a partir de uma matriz preexistente utilizando o método *arma::span()* criando nova matriz que contem os três últimos elementos da primeira coluna de *armaSV*.

```

1     void LQR::computeControl()
2     {
3         // Calcular sinal de referencia
4         computeRef();
5
6         // Create local variable for error
7         arma::mat errL;
8
9         // Reference-actual
10        errL = armaRef - armaSV(arma::span(9, 11), 0);
11
12        // First loop LQR
13        errL = armaK1 * errL;
14

```

```

15         // Integrate error
16         armaError = armaError + errL * dt;
17
18         // Second loop LQR
19         armaOutputControl = armaKx * armaSV;
20
21         // Actuator Output
22         armaOutputControl = armaError - armaOutputControl + armaTrim;
23
24         // Para enviar os dados para o MATLAB estes devem estar vetorizados
25         vetorizarOutControl();
26     }

```

O procedimento que ocorre em (8) é a invocação da função *vetorizarOutControl()* que realiza adequação dos dados da matriz 4x1 que contem as velocidades, em rad/s, calculadas (*armaOutputControl*) para um vetor de quatro posições (*OutputControlVetorizado*). O mesmo é feito da seguinte maneira:

```

1     void LQR::vetorizarOutControl()
2     {
3         // Converter para vetor para enviar para o MATLAB
4         for(size_t i = 0; i < iOutputControl; i++)
5         {
6             for(size_t j = 0; j < jOutputControl; j++)
7             {
8                 OutputControlVetorizado[i] = arma::as_scalar( ...
9                     armaOutputControl(i, j) );
10            }
11        }

```

Os dados estão armazenados e posteriormente serão recuperados no código fonte principal. Após o fim de *vetorizarOutControl()* volta-se a *computeControl()* e em (10) os procedimentos de cálculo são finalizados.

5.4.4 Obter as saídas da lei de controle

Para se obter as saídas no código fonte principal, deve-se utilizar o método *getOutputControl()* conforme o fragmento de código mostrado:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              for(int i = 0; i < numeroIteracoes; i++)
8              {
9                  //(...)
10                 // Obter saída do controlador
11                 ptrOutCntToMatlab = lqr.getOutputControl();
12                 //(...)
13             }
14             //(...)
15         }
16         return 0;
17     }

```

A função retorna o endereço de memória do vetor onde foram armazenadas as saídas de forma interna na classe. O endereço é armazenado em *ptrOutCntToMatlab*, um ponteiro do código fonte principal. A atribuição do endereço é realizada da seguinte forma:

```

1      double* LQR::getOutputControl()
2      {
3          return OutputControlVetorizado;
4      }

```

A partir deste ponto, as saídas estão disponíveis para serem utilizadas no escopo de *main.cpp*

5.4.5 Retornar dimensão, em bytes, do vetor das saídas do controlador

O último uso de um método da classe no código fonte principal é no momento de obter a dimensão do vetor para enviar as saídas, armazenadas em *ptrOutCntToMatlab*, para o MATLAB®. Este uso ocorre no fragmento abaixo, utilizando-se *lqr.getOutputCtlSize()* como argumento:

```

1      int main()
2      {
3          //(...)
4          while(true)
5          {
6              //(...)
7              for(int i = 0; i < numeroIteracoes; i++)
8              {
9                  //(...)
10                 comm.enviarDados(ptrOutCntToMatlab, ...
11                                 lqr.getOutputCtlSize());
12             }
13             //(...)
14         }
15         return 0;
16     }

```

Esta função é implementada da seguinte forma:

```

1      size_t LQR::getOutputCtlSize()
2      {
3          sizeOutCtlVet = sizeof(OutputControlVetorizado);
4          return sizeOutCtlVet;
5      }

```

5.5 Classe Motor

Os códigos que lidam e implementam as funcionalidades dos motores é a classe Motor definida nos arquivos *Motor.hpp* e *Motor.cpp*. Nesta Seção serão discutidas estas implementações.

No código fonte principal a classe Motor é utilizada através dos procedimentos listados a seguir. Durante esta Seção serão discutidas as implementações destes processos e indicado onde cada procedimento é utilizado em *main.cpp*

- Instanciar objeto da classe Motor
- Converter velocidades em *duty cycles* equivalentes
- Estabelecer valor de *duty cycle* nos motores
- Encerrar rotação dos motores

5.5.1 Instanciar objeto da classe Motor

No código fonte principal a classe é utilizada pela primeira vez para criar objetos referentes a cada um dos quatro motores. São atribuídos como argumentos para o construtor dos objetos o valor do sentido de rotação que pode ser +1 ou -1 e em seguida o caminho para o arquivo de *duty cycle* no sistema operacional do microcontrolador BeagleBone Green.

```
1  int main()
2  {
3      //(...)
4      Motor motor1(default_sentidoRotacaoM1, default_M1.path),
5              motor2(default_sentidoRotacaoM2, default_M2.path),
6              motor3(default_sentidoRotacaoM3, default_M3.path),
7              motor4(default_sentidoRotacaoM4, default_M4.path);
8      //(...)
9      return 0;
10 }
```

No fragmento de código acima, pode-se ver o uso de diretivas que são definidas em *Motor.hpp* da seguinte forma:

```

1 // Caminho para os arquivos no sistema operacional
2 #define default_M1_path "/sys/class/pwm/pwmchip0/pwm0/duty-cycle"
3 #define default_M2_path "/sys/class/pwm/pwmchip3/pwm0/duty-cycle"
4 #define default_M3_path "/sys/class/pwm/pwmchip1/pwm1/duty-cycle"
5 #define default_M4_path "/sys/class/pwm/pwmchip1/pwm0/duty-cycle"
6
7 // Sentido de rotacao do motor
8 #define default_sentidoRotacaoM1 -1
9 #define default_sentidoRotacaoM2 -1
10 #define default_sentidoRotacaoM3 +1
11 #define default_sentidoRotacaoM4 +1

```

Internamente na classe, o construtor atribui o valor do sentido de rotação (*sentidoRotacao*) para a variável membra da classe *sentidoRotacao_*. O mesmo ocorre para *string motorFilepath* que será armazenada em *pathM_*. Estes dados são parâmetros cruciais para o uso de cada motor, então eles são armazenados internamente em cada objeto para serem utilizados posteriormente. *sentidoRotacao_* será utilizado para verificar se o sinal da velocidade calculada pelo controlador está correto para determinado motor antes que este dado seja enviado como comando para o motor. *pathM_* será utilizado para especificar qual arquivo abrir para escrever o valor do *duty cycle*.

Após armazenar os dados, são imprimidas mensagens na tela para que o usuário possa rever quais valores foram atribuídos como argumentos para o construtor da classe.

```

1 Motor::Motor(const int sentidoRotacao, const char* motorFilepath) :
2     sentidoRotacao_(sentidoRotacao),
3     pathM_(motorFilepath)
4 {
5     std::cout << "Motor criado: " << std::endl;
6     std::cout << "Sentido de Rotacao: " << sentidoRotacao_ << ...
7     std::endl;

```

```

7         std::cout << "Filepath motor: " << pathM_ << std::endl;
8         std::cout << "-----" << std::endl;
9     }

```

5.5.2 Converter velocidades em duty cycles equivalentes

O próximo momento onde os métodos da classe são utilizados em *main.cpp* é após o controlador calcular as velocidades de saída em rad/s. Neste ponto são aplicados os métodos para converter estes valores para os *duty cycles* equivalentes através da função *convertVelocity_PWM()* sendo utilizada no código fonte principal da seguinte forma:

```

1     int main()
2     {
3         //(...)
4         while(true)
5         {
6             //(...)
7             for(int i = 0; i < numeroIteracoes; i++)
8             {
9                 //(...)
10                // Obtem a saida do controlador e atribui como ...
11                // argumento para os metodos abaixo
12                motor1.convertVelocity_PWM(ptrOutCntToMatlab[1]);
13                motor2.convertVelocity_PWM(ptrOutCntToMatlab[3]);
14                motor3.convertVelocity_PWM(ptrOutCntToMatlab[0]);
15                motor4.convertVelocity_PWM(ptrOutCntToMatlab[2]);
16                //(...)
17            }
18        }
19        //(...)
20        return 0;
21    }

```

Conforme descrito na Seção 5.4, referente à classe LQR, as saídas armazenadas em *ptrOutCntToMatlab* são utilizadas como argumentos para *convertVelocity_PWM()* que multiplica estes dados de *input* (*velocity*) por uma constante a fim de se obter a equivalência entre rad/s e *duty cycle*. O resultado é armazenado na variável interna *PWM_* que será utilizada em outro método para escrever no arquivo de cada motor.

Note que a variável *PWM* é do tipo inteiro. Então, o resultado do produto da velocidade e da constante será truncado e armazenado como um valor inteiro.

A equivalência entre o *duty cycle* e a velocidade angular foi obtida de forma experimental através de bancada de testes. Como segunda observação, destaca-se que a zona morta não é considerada.

```

1      int Motor::convertVelocity_PWM(double velocity)
2      {
3          // 9.54/103*10000 e uma constante responsavel pela equivalencia ...
           entre rad/s e duty cycle
4          int PWM = velocity * (9.54/103*10000);
5          PWM_ = PWM;
6          return PWM; //Pode ser utilizada opcionalmente para verificacao
7      }

```

5.5.3 Estabelecer valor de *duty cycle* nos motores

Depois da conversão, o código principal utiliza o método *setPWM()* para estabelecer os dados de *duty cycle* nos motores. Este método é detalhado no fluxograma da Fig. 5.8.

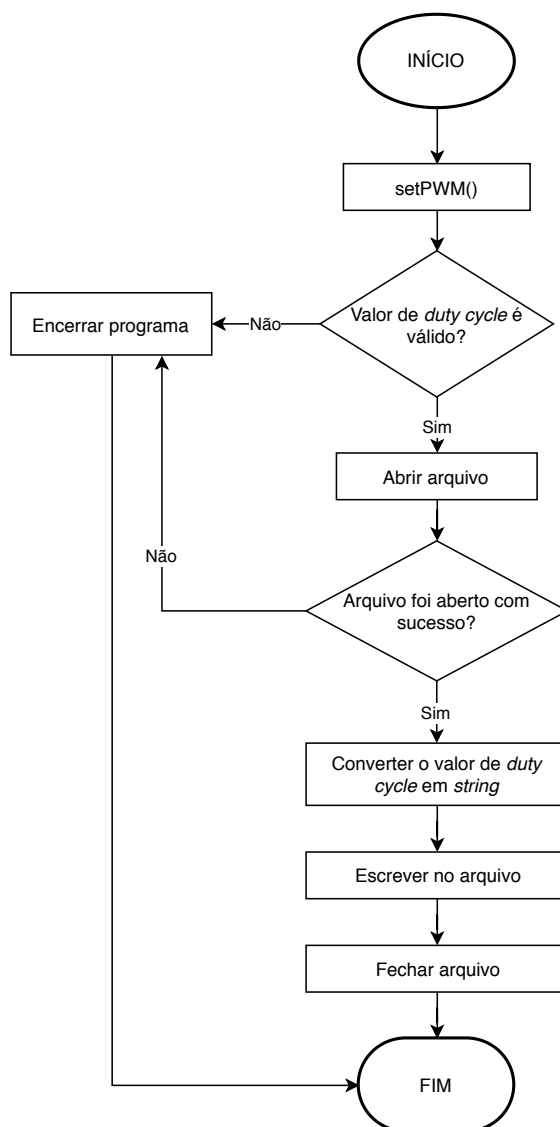


Figura 5.8 – Fluxograma dos procedimentos internos que ocorrem no método *setPWM()* da classe Motor.

O primeiro procedimento ocorre quando *setPWM()* é invocada no código fonte principal, conforme ilustrado no fluxograma da Fig. 5.8.

```

1   int main()
2   {
3       //(...)
4       while(true)
5       {
6           //(...)
7           for(int i = 0; i < numeroIteracoes; i++)

```

```

8      {
9          //(...)
10         // Estabelecer duty cycle nos motores
11         motor1.setPWM();
12         motor2.setPWM();
13         motor3.setPWM();
14         motor4.setPWM();
15         //(...)
16     }
17 }
18 //(...)
19 return 0;
20 }

```

Após ser invocado, o método declara uma variável (*file*) para receber o descritor de arquivo em um momento posterior quando o arquivo de *duty cycle* for aberto.

Na sequência, é utilizado um mecanismo para garantir que determinado rotor gire em um sentido específico. Isso deve ser realizado pois a hélice acoplada no rotor deve rotacionar apenas no sentido estabelecido conforme a fabricação da mesma. Então, para se alcançar este objetivo, o resultado da conversão armazenada em *PWM_* é multiplicado pelo sentido de rotação. Se este produto for menor que zero, significa que o sinal da velocidade calculada para este motor é diferente do sentido de rotação que ele deve apresentar e desta forma, para evitar este efeito indesejado, é impressa mensagem notificando o usuário, o motor é parado e o programa é encerrado. Outra situação que desencadeia o resultado descrito anteriormente é quando o valor calculado para o *PWM_* é superior que o período estabelecido para o sinal PWM.

O valor do período utilizado no projeto é definido em *Motor.hpp*:

```

1 // Valor padrao utilizado no projeto em nanosegundos
2 #define default_period.BBG 3000000 // 3 ms

```

Após realizar esta verificação, o arquivo de *duty cycle* será aberto, com base no caminho especificado na construção do objeto, e será realizada confirmação se houve sucesso em abrir o

mesmo utilizando o retorno da *system call open()*. Caso seja retornado -1 para o descritor de arquivo, significa que houve erro, logo, o programa será encerrado.

O valor que será escrito no arquivo deve ser um inteiro positivo convertido em uma *string*. Então, como primeiro passo, declara-se uma variável para armazenar esta *string* (*string_PWM*). O segundo passo é realizar a conversão do inteiro para *string*, o mesmo ocorre através do uso da função *sprintf()* atribuindo-lhe como argumentos o endereço da variável que irá armazenar o resultado da conversão, o formato do dado que será usado para a conversão ("%d" define que este dado trará-se de um inteiro) e, como último argumento, o valor do dado.

Deve-se realizar destaque sobre a lógica para se obter o valor de *duty cycle* que é a seguinte: o primeiro ponto é, realizar o produto entre *PWM_* e *sentidoRotacao_*. Esta operação é realizada para obter o módulo, uma vez que o valor de *duty cycle* a ser escrito no arquivo deve ser um inteiro positivo. Como a conversão da velocidade para o equivalente em *duty cycle* não altera o sinal deste primeiro termo, torna-se necessário neste momento obter o módulo desta conversão, logo, para os motores que devem ter velocidades negativas, na multiplicação pelo sentido de rotação também negativo será obtido o módulo da conversão. O segundo ponto a ser discutido sobre esta lógica, é a soma realizada entre o produto e a diretiva *default_dutyCycle_BBG*, definida em *Motor.hpp*:

```
1 // Valor padrao utilizado no projeto em nanosegundos
2 #define default_dutyCycle_BBG 1000000 // 1 ms
```

É realizada esta soma pois deseja-se que o pulso, relativo ao PWM, apresente nível lógico alto durante um intervalo de tempo entre 1 ms a 2 ms. Desta forma, é garantido a obtenção de um intervalo de duração mínima de 1 ms. Conforme explicado na Seção 5.1 este intervalo de tempo do pulso é definido como o *duty cycle* no domínio do microcontrolador BeagleBone.

Após converter e armazenar o dado, será escrito no arquivo o dado relativo ao *duty cycle* através da *system call write()*. Para realizar o mesmo, são atribuídos como argumentos o descritor de arquivo, *file*, que indica em qual arquivo será escrita a informação. O segundo argumento é o endereço da variável que armazena a *string* a ser escrita. O último argumento é a quantidade de *bytes* que serão escritos.

Finalizado o processo de escrita, o arquivo é fechado utilizando-se a *system call close()* atribuindo-lhe como argumento o descritor de arquivo. Após estes procedimentos o comando de estabelecer o *duty cycle* é finalizado e, caso não ocorra erro, o motor terá velocidade equivalente a calculada pelo controlador.

```

1      void Motor::setPWM()
2      {
3          int file;
4
5          // Checar resultados de entrada da funcao
6          if(PWM_ * sentidoRotacao_ < 0 || PWM_ * sentidoRotacao_ > ...
            default_period_BBG))
7      {
8          std::cout << "Erro: setPWM: valores invalidos de pwm ou ...
                sentido de rotacao!" << std::endl;
9
10         //Parar motor -- saturar em 0
11         pararMotor();
12
13         // Encerrar programa
14         exit(EXIT_FAILURE);
15     }
16
17     file = open(pathM_, O_RDWR | O_CREAT);
18
19     // Checar se abriu o arquivo do motor com sucesso
20     if(file < 0)
21     {
22         std::cout << "Erro: setPWM: fracasso ao abrir o arquivo de ...
                duty_cycle do motor!" << std::endl;
23         exit(EXIT_FAILURE);
24     }
25
26     char string_PWM[32];
27     sprintf(string_PWM, "%d", PWM_ * sentidoRotacao_ + ...

```

```

        default_dutyCycle_BBG);
28
29     write(file, string_PWM, sizeof(string_PWM));
30     close(file);
31 }

```

5.5.4 Encerrar rotação dos motores

O último uso da classe, na execução de uma iteração do laço *while(true)*, ocorre após finalizar as execuções previstas pelo número de iterações estabelecidas para o *hardware-in-the-loop*. Este evento ocorre em *main.cpp* no fragmento a seguir:

```

1     int main()
2     {
3         //(...)
4         while(true)
5         {
6             //(...)
7             for(int i = 0; i < numeroIteracoes; i++)
8             {
9                 //(...)
10            }
11            motor1.pararMotor();
12            motor2.pararMotor();
13            motor3.pararMotor();
14            motor4.pararMotor();
15        }
16        //(...)
17        return 0;
18    }

```

A finalidade desta função é encerrar o movimento de rotação dos motores quando finalizar a execução do *hardware-in-the-loop*. Para alcançar este objetivo a função *pararMotor()*

invoca o método *setPWM()* atribuindo-lhe como argumento o resultado da função *convertVelocity_PWM()* que por sua vez recebe como argumento um valor de velocidade igual a zero. Desta forma este valor será convertido para um valor de *duty cycle* que implica em velocidade nula dos motores. O fluxograma da Fig. 5.9 mostra de forma visual este procedimento.

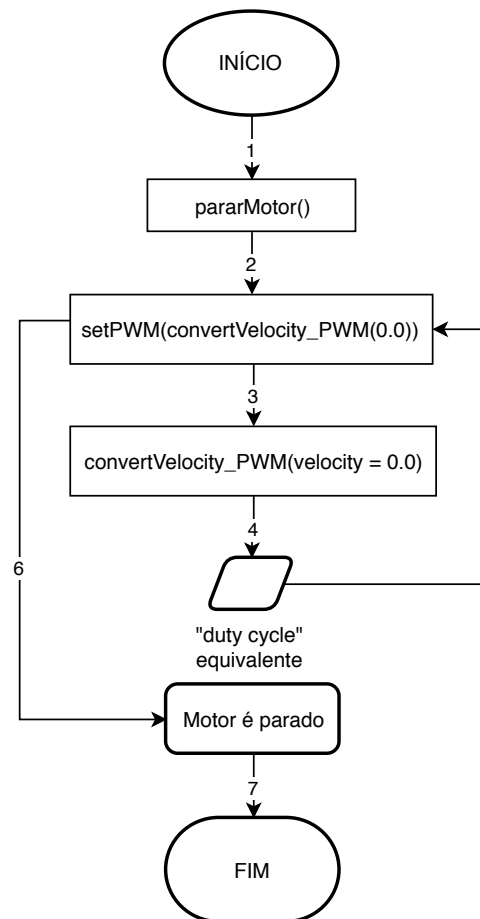


Figura 5.9 – Fluxograma que ilustra os processos que ocorrem para estabelecer velocidade nula nos motores.

Em (1) é o momento quando o método é invocado no código fonte principal. O próximo procedimento (2) ocorre na definição do método *pararMotor()* dentro da classe onde é invocada a função *setPWM()* atribuindo-lhe outra função:

```

1      void Motor::pararMotor()
2      {
3          setPWM(convertVelocity_PWM(0.0));
4      }
  
```

Em (3) é ilustrado o procedimento de invocar *convertVelocity_PWM()* onde lhe é atribuído o argumento zero o que implicará em converter uma velocidade nula para um *duty cycle* equivalente. É representado em (4) a saída calculada pela função *convertVelocity_PWM()*. (5) representa este dado calculado sendo a entrada da função *setPWM()*. O procedimento (6) representa a saída produzida pela função que resultará no fim da movimentação dos rotores.

5.6 Função de emergência *Emergency*

Nesta Seção será discutida a implementação da função de emergência, definida nos arquivos *Emergency.hpp* e *Emergency.cpp*, onde, através do fluxograma, da Fig. 5.10 pode-se ver seu funcionamento.

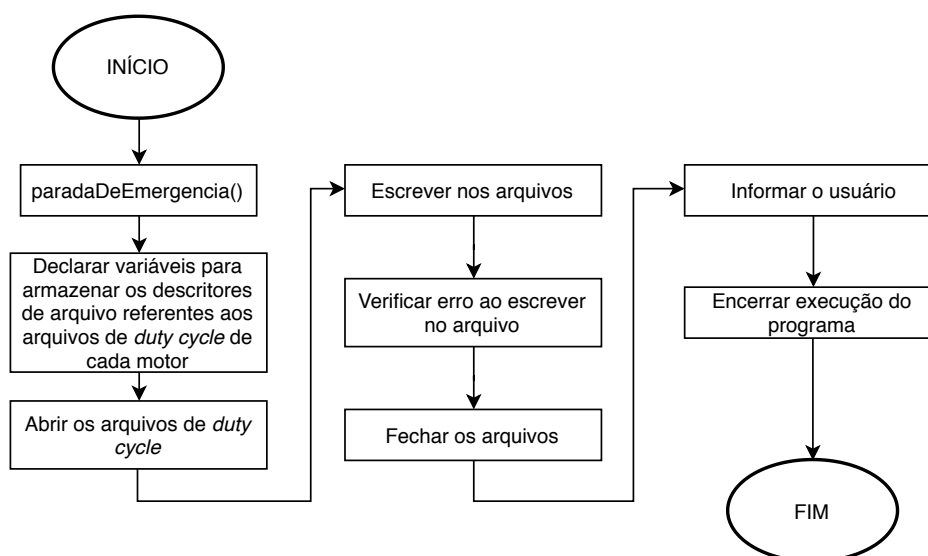


Figura 5.10 – Fluxograma dos processos que ocorrem internamente na função de emergência quando esta é invocada.

5.6.1 *paradaDeEmergencia()*

Conforme descrito ao longo da Seção 5.2, a função é inicializada em *main.cpp*, conforme fragmento abaixo, onde começa o processo de monitoramento que verifica se as teclas *ctrl + c* são pressionadas, caso elas sejam, a função de emergência será invocada.

```

1     int main()
2     {
3         //(...)
4         // Parada de emergencia - ctrl + c
5         signal(SIGINT, paradaDeEmergencia);
6         //(...)
7         return 0;
8     }

```

5.6.2 Declarar variáveis

Quando a função entrar em execução, o primeiro procedimento é declarar variáveis para armazenarem os descritores de arquivo.

```

1     void paradaDeEmergencia(int sig)
2     {
3         int fileM1, fileM2, fileM3, fileM4;
4         //(...)
5     }

```

5.6.3 Abrir os arquivos

Após as declarações, o processo de abrir os arquivos de *duty cycle* é iniciado.

```

1     void paradaDeEmergencia(int sig)
2     {
3         //(...)
4         fileM1 = open(default.M1_path, O_RDWR | O_CREAT);
5         fileM2 = open(default.M2_path, O_RDWR | O_CREAT);
6         fileM3 = open(default.M3_path, O_RDWR | O_CREAT);
7         fileM4 = open(default.M4_path, O_RDWR | O_CREAT);

```

```

8         // (...)
9     }

```

Para abrir cada arquivo é utilizado a *system call*: *open* atribuindo-lhe como argumentos uma *string* que é o caminho dentro do sistema operacional do microcontrolador para o arquivo de *duty cycle* e dois modos de acesso: *O_RDWR* e *O_CREAT*. O primeiro permite a leitura e escrita no arquivo existente. Já o segundo, cria o arquivo caso ele não exista para que possa ser aberto. O retorno da *system call* é o descritor de arquivo daquele que foi aberto.

Os argumentos de *open()* referentes aos caminhos dos arquivos são diretivas definidas em *Motor.hpp*. Como por exemplo: *default.M2_path* que lhe é atribuída a *string*: “/sys/class/pwm/pwmchip3/pwm0/duty_cycle”.

5.6.4 Escrever nos arquivos

Após abrir os arquivos, será escrito nestes um valor de *duty cycle* que resulta em uma velocidade angular nula nos rotores. Para escrever nestes, utiliza-se *sprintf()* e em seguida a *system call write()*.

A função *sprintf()*, implementada na biblioteca *stdio.h*, é utilizada para escrever em uma variável uma *string* que contem determinado dado formatado. O dado formatado em questão é um dado do tipo *int* que é o valor de *duty cycle*. É necessário o uso da função *sprintf()* pois deve-se escrever uma *string* no arquivo de *duty cycle*. Então esta realiza a conversão do dado inteiro na para uma *string*.

Depois da conversão, a *string*, será escrita no arquivo utilizando-se *write()*. Atribuindo-lhe como argumento o descritor de arquivo, a *string* que representa o valor de *duty cycle* e a quantidade de *bytes* que serão escritos no arquivo.

5.6.5 Verificar erro e fechar os arquivos

A verificação de erro ao escrever nos arquivos é feita através do retorno de *write()* que retorna -1 quando ocorre erro. A função é utilizada para escrever o dado, caso execute com sucesso a mensagem de erro não será exibida e o arquivo será fechado, caso contrário será

mostrada mensagem indicando em qual arquivo houve erro ao escrever. O fragmento de código mostra os procedimentos descritos.

```
1  void paradaDeEmergencia(int sig)
2  {
3      //(...)
4      char string_PWM_M1[20], string_PWM_M2[20], string_PWM_M3[20], ...
          string_PWM_M4[20];
5      // Armazenar dados como string
6      sprintf(string_PWM_M1, "%d", default_dutyCycle_BBG);
7      sprintf(string_PWM_M2, "%d", default_dutyCycle_BBG);
8      sprintf(string_PWM_M3, "%d", default_dutyCycle_BBG);
9      sprintf(string_PWM_M4, "%d", default_dutyCycle_BBG);
10     // Escrever nos arquivos, verificar erro e fechar os arquivos
11     if(write(fileM1, string_PWM_M1, sizeof(string_PWM_M1)) < 0)
12     {
13         std::cout << "Erro na parada de emergencia. Erro ao ...
                escrever no motor1!" << std::endl;
14     }
15     close(fileM1);
16
17     if(write(fileM2, string_PWM_M2, sizeof(string_PWM_M2)) < 0)
18     {
19         std::cout << "Erro na parada de emergencia. Erro ao ...
                escrever no motor2!" << std::endl;
20     }
21     close(fileM2);
22
23     if(write(fileM3, string_PWM_M3, sizeof(string_PWM_M3)) < 0)
24     {
25         std::cout << "Erro na parada de emergencia. Erro ao ...
                escrever no motor3!" << std::endl;
26     }
27     close(fileM3);
28
```

```

29     if(write(fileM4, string_PWM_M4, sizeof(string_PWM_M4)) < 0)
30     {
31         std::cout << "Erro na parada de emergencia. Erro ao ...
                escrever no motor4!" << std::endl;
32     }
33     close(fileM4);
34     //(...)
35 }

```

5.6.6 Informar o usuário

Após estes procedimentos, é exibida mensagem para o usuário informando que a função foi invocada devido ao uso das teclas *ctrl + c* e que os rotores foram parados.

```

1     void paradaDeEmergencia(int sig)
2     {
3         //(...)
4         std::cout << ...
                "-----" << ...
                std::endl;
5         std::cout << "Ctrl+C pressionado! Parada de emergencia!" << ...
                std::endl;
6         std::cout << "Parada de emergencia. Parada a rotacao dos ...
                motores." << std::endl;
7         std::cout << ...
                "-----" << ...
                std::endl;
8         //(...)
9     }

```

5.6.7 Encerrar execução do programa

O último passo que finalizará a função de emergência é o uso da *system call* `exit()` que encerra a execução do programa quando é invocada. Ela pode ser utilizada para sinalizar se o processo no sistema operacional terminou com ou sem sucesso através das constantes `EXIT_SUCCESS` e `EXIT_FAILURE` que podem ser atribuídas como argumento para a função. No caso da função de emergência, é utilizado `EXIT_FAILURE` para indicar que o programa finalizou de forma não esperada.

```
1  void paradaDeEmergencia(int sig)
2  {
3      //(...)
4      // Encerra a execucao do processo
5      exit(EXIT_FAILURE);
6  }
```

CAPÍTULO VI

RESULTADOS E DISCUSSÕES

Nesta Capítulo serão apresentados resultados de simulação realizada utilizando a técnica HIL baseada na comunicação entre MATLAB[®] e microcontrolador descrita nos Capítulos IV e V.

6.1 Simulações

A simulação realizada que gerou os resultados que serão mostrados envolve:

1. Energizar o sistema VANT: microcontrolador e atuadores
2. Colocar o servidor em execução
3. Configurar no cliente (MATLAB[®]) a referência como um sinal do tipo degrau unitário para o controlador
4. Executar rotina cliente para se conectar com o servidor e iniciar o *hardware-in-the-loop*
5. Finalizadas as iterações, utilizar os resultados obtidos para análise na rotina cliente

6.2 Resultados e análises

Os resultados de simulação foram obtidos para uma referência do tipo degrau unitário em cada eixo (x_e, y_e, z_e) . O tempo de simulação teve duração de 60 s. Os dados que serão

analisados foram obtidos através da rotina no MATLAB® referente ao cliente do sistema sendo estes dados relacionados a velocidade dos motores, coordenadas de posição, velocidade linear, angular e ângulos de Euler do centro de gravidade do VANT em relação ao tempo decorrido de uso do sistema. Também são obtidos e analisados os dados relativos a amostragem, sendo estes relacionados ao tempo de solução da ODE, responsável por calcular o estado da planta, ao tempo necessário para realizar a comunicação servidor-cliente e computar a ação de controle.

Pode-se observar nas Figs. 6.1 e 6.2 as velocidades que os motores do multirrotor apresentaram durante os primeiros instantes de execução do sistema. As magnitudes das velocidades dos motores são relativamente semelhantes, apesar dos sinais diferentes. Os motores 1 e 2 apresentam velocidades negativas e os motores 3 e 4, positivas.

Ao analisar as Figs. 6.1 e 6.2 pode-se perceber que as variações de velocidades ocorrem durante os primeiros 15 s de execução do sistema. Na Fig. 6.1 observa-se que o máximo valor da velocidade para o motor 3 é aproximadamente, 202 rad/s no instante de tempo 1 s. Já o motor 4: 201 rad/s em 1 s também. Os valores mínimos para os motores 3 e 4 são, aproximadamente, 197,75 rad/s em 4 s. Já o valor de velocidade que se mantém constante até o final da simulação, para ambos motores, é, aproximadamente, 199 rad/s e inicia-se em 14 s.

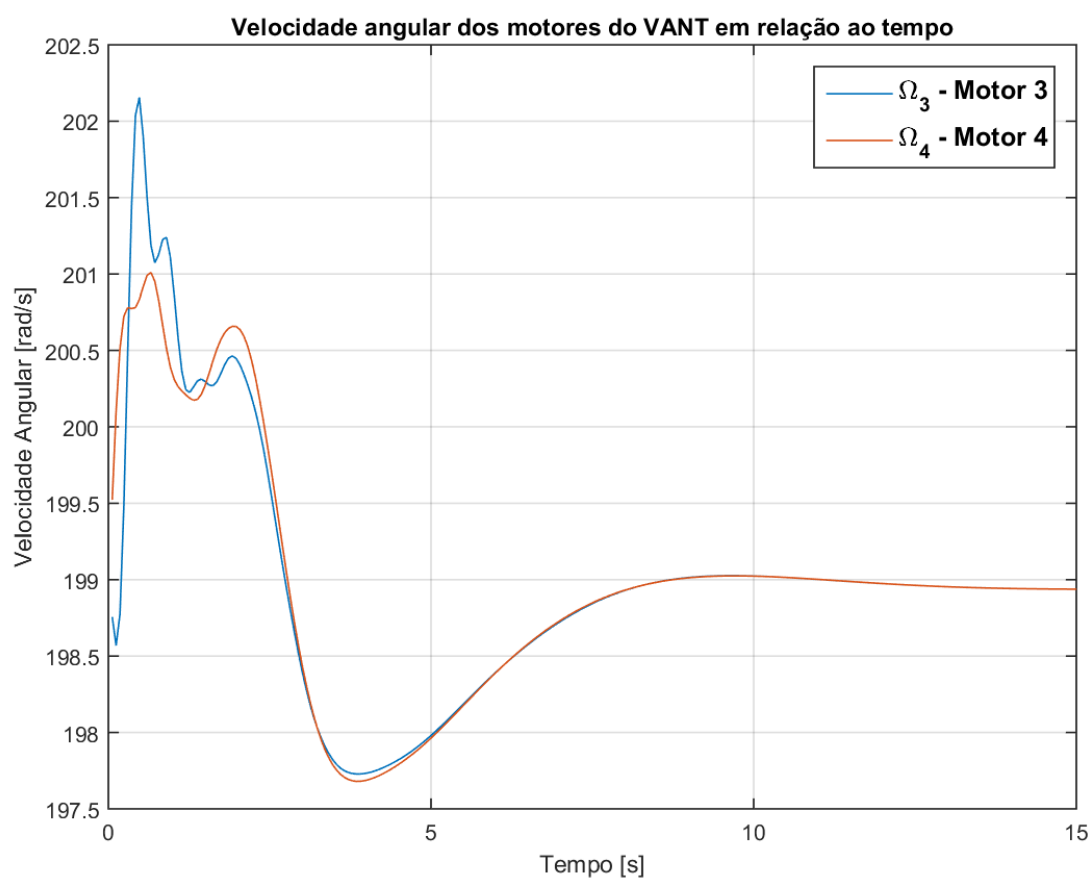


Figura 6.1 – Velocidade angular dos motores 3 e 4 do VANT, em relação ao tempo, com mais detalhes.

Na Fig. 6.2 pode-se visualizar os valores de máximo, mínimo e contante e os instantes que ocorrem. As máximas velocidades dos motores 1 e 2 são, aproximadamente, -197,75 rad/s e ocorrem no instante 4 s. Já as mínimas velocidades de ambos motores são, aproximadamente, -201,25 rad/s ocorrendo em 0,5 s. As velocidades constantes dos dois motores são, aproximadamente, -199 rad/s e ocorrem a partir do instante 14 s.

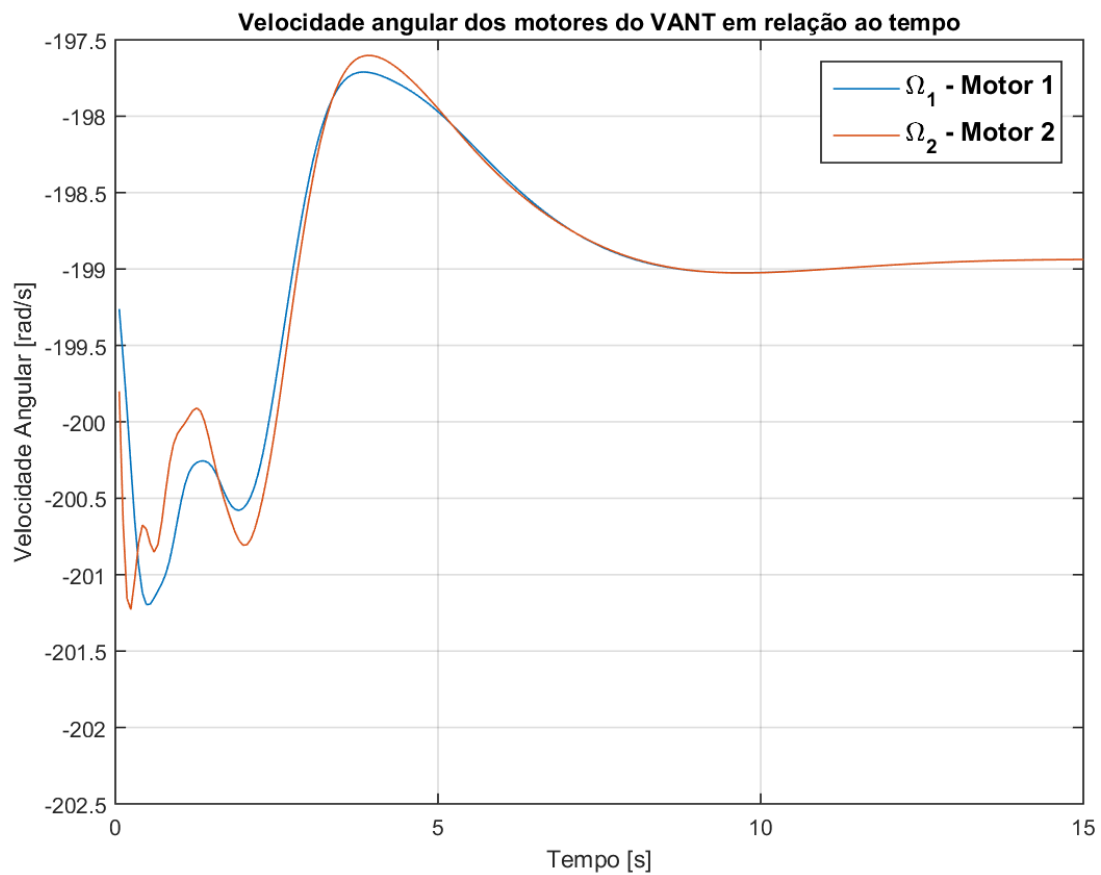


Figura 6.2 – Velocidade angular dos motores 1 e 2 do VANT, em relação ao tempo, com mais detalhes.

A análise dos resultados de posição do CG do VANT, em relação ao tempo, pode ser realizada através da Fig. 6.3. Nos três eixos ocorreram *overshoots* em relação a referência. No eixo z_e ocorreu maior *overshoot* em um instante de tempo posterior aos que ocorrem nos eixos x_e e y_e . Também pode-se observar que as intensidades de *overshoot* nestes dois últimos eixos são semelhantes. Outro aspecto que pode ser observado é a convergência do sistema nos três eixos. Em x_e e y_e a convergência ocorre, aproximadamente, após 10 s. Já em z_e , 16 s.

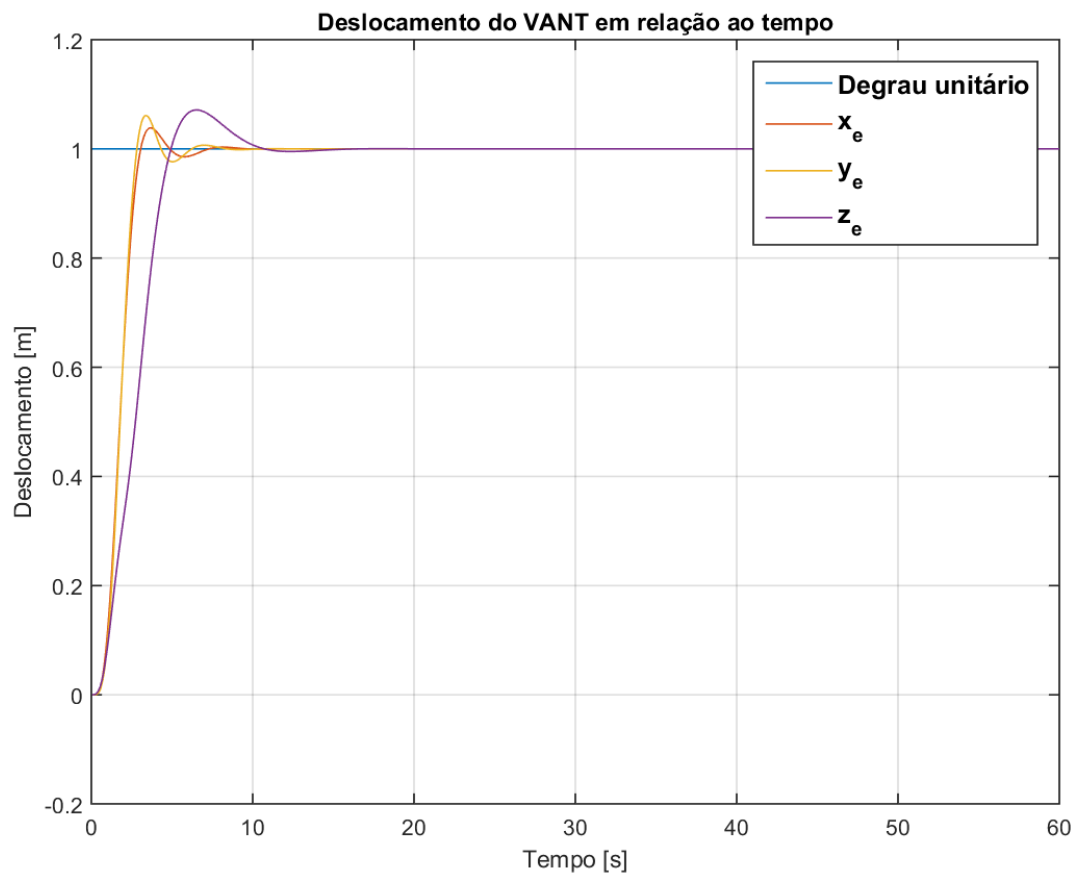


Figura 6.3 – Posição do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.

Na Fig. 6.4 pode-se ver com mais detalhes os resultados relativos aos *overshoots* e convergência do sistema.

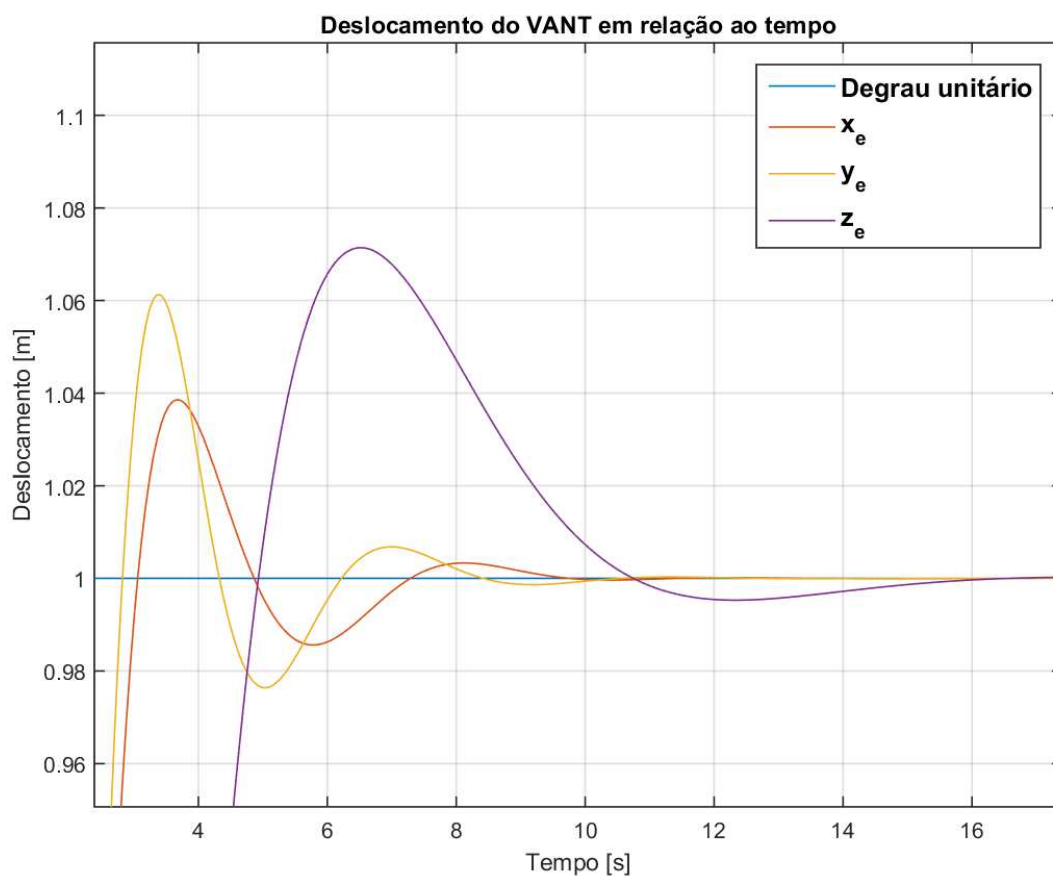


Figura 6.4 – Posição do CG do VANT, em relação ao tempo, com mais detalhes.

A Fig. 6.5 mostra as velocidades lineares do CG do VANT durante a execução dos procedimentos. Através dela percebe-se que existem oscilações nas velocidades durante, aproximadamente, os primeiros 15 s. Após este intervalo todas as três velocidades são nulas uma vez que a planta atingiu o objetivo estabelecido pela referência e permanece em repouso neste ponto.

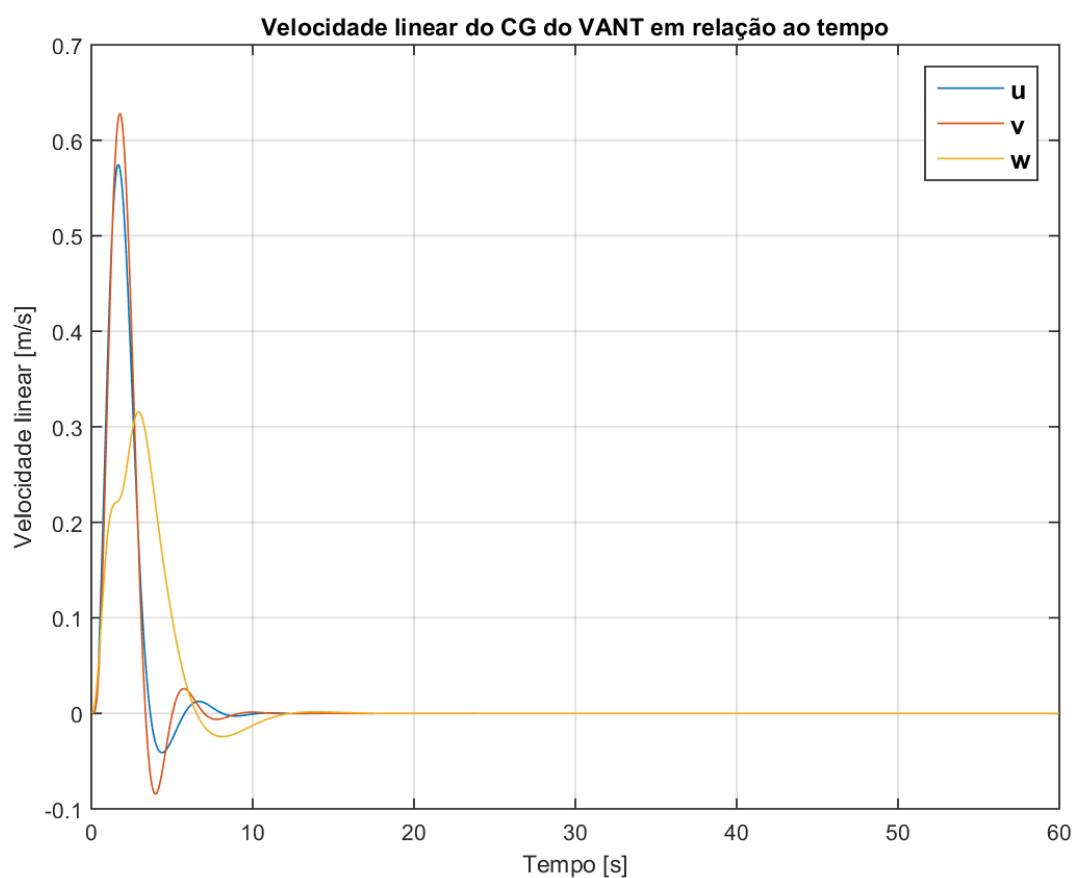


Figura 6.5 – Velocidade linear do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.

Através da Fig. 6.6 pode-se visualizar as velocidades angulares do CG do VANT ao longo do tempo. De forma semelhante aos dados das velocidades lineares, os valores das velocidades angulares oscilam até, aproximadamente, os primeiros 10 s de simulação. Deste ponto em diante estas apresentam valor nulo.

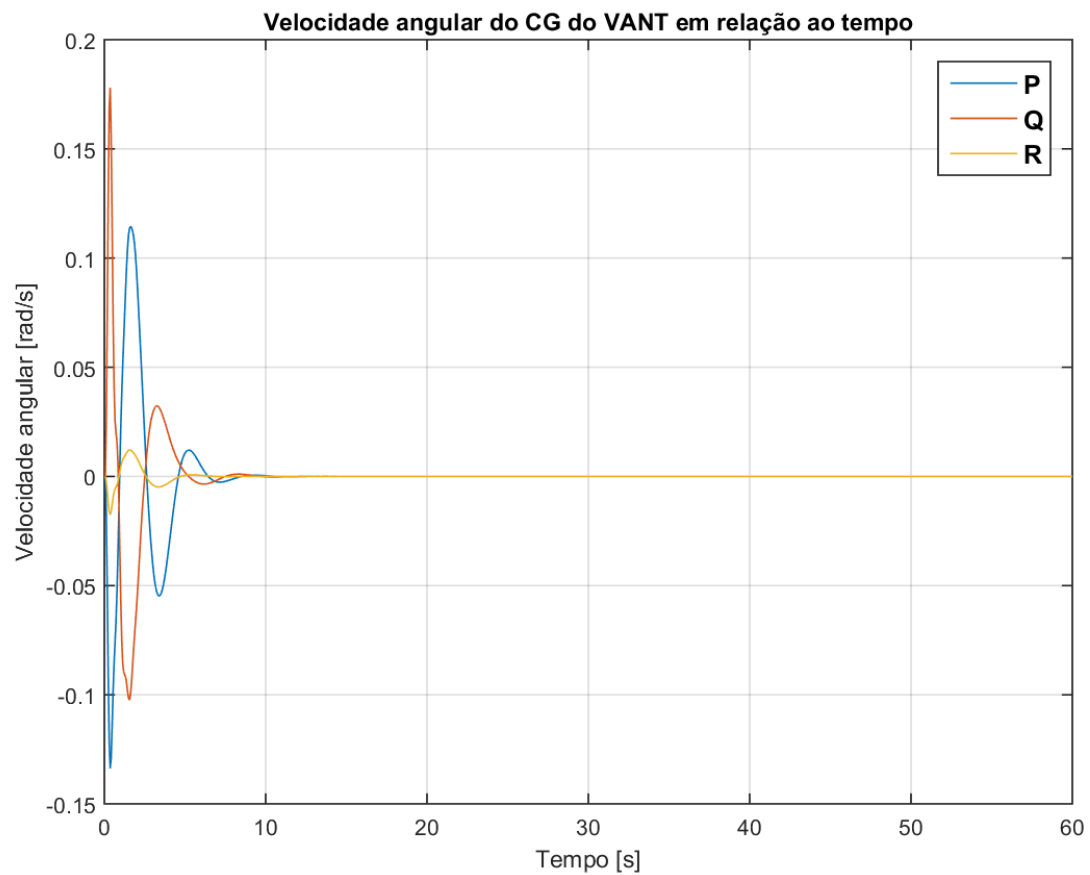


Figura 6.6 – Velocidade angular do CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.

Na Fig. 6.7 são mostrados os ângulos de Euler do CG do VANT em função do tempo. Pode-se perceber através da mesma que os ângulos de arfagem (ϕ) e rolagem (θ) apresentam maiores variações em relação ao ângulo de guinada (ψ). Com exceção da guinada, os ângulos de Euler convergem para zero. Segundo Marques (2018), o ângulo de guinada não converge para zero neste caso devido a eficiência da resposta do sistema devido ao arrasto das hélices.

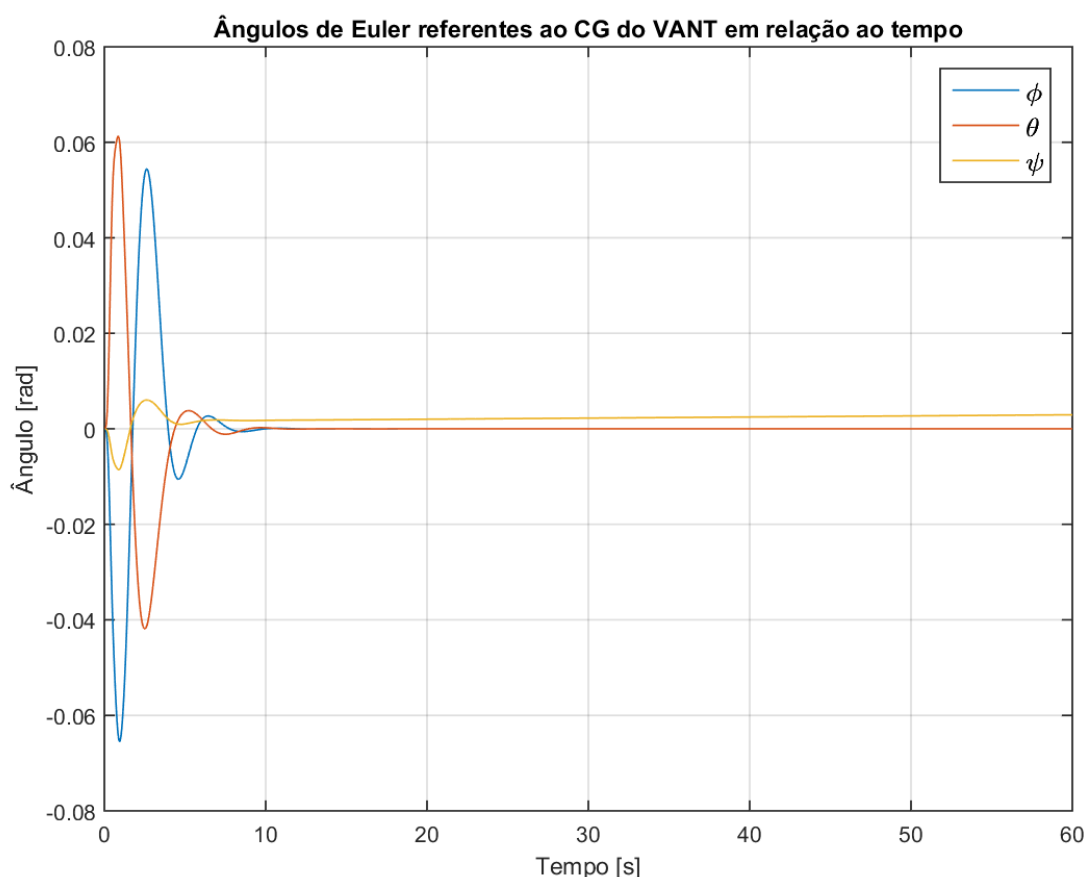


Figura 6.7 – Ângulos de Euler referentes ao CG do VANT, em relação ao tempo, submetido a uma entrada degrau unitário.

Por fim, o tempo de amostragem do sistema é analisado a fim de verificar se este é suficiente para que os procedimentos de calcular o estado da planta, através da solução da ODE, realizar a comunicação entre servidor-cliente e para computar a lei de controle, possam ser concluídos. A análise da Fig. 6.8 revela que o tempo necessário para a solução da ODE é superior que o tempo exigido para realizar a comunicação entre os sistemas e o tempo para computar a lei de controle.

A partir dos dados, exibidos em Fig. 6.8, pode-se obter a média e o desvio padrão do tempo de cálculo do estado da planta equivalentes a 0,0178 s e 0,0098 s, respectivamente. Para o tempo para comunicar e computar as saídas da lei de controle, obtêm-se média e desvio padrão equivalentes a 0,0024 s e 0,0011 s, respectivamente.

Analizando a Fig. 6.8 e as médias pode-se concluir que o tempo de amostragem de 0,060 s é suficiente, pois é superior ao tempo total exigido para calcular o estado da planta,

realizar comunicação entre os sistemas e calcular as ações de controle.

A análise destes dados é realizada sobre os tempos obtidos na rotina no MATLAB® em cada iteração durante a execução do *hardware-in-the-loop*. Desta forma o tempo de comunicação relativo ao envio dos parâmetros iniciais de controle pela rotina cliente não é obtido para análise, uma vez que, a simulação HIL não se iniciou neste momento e, nem sequer, os atuadores do VANT são comandados neste ponto.

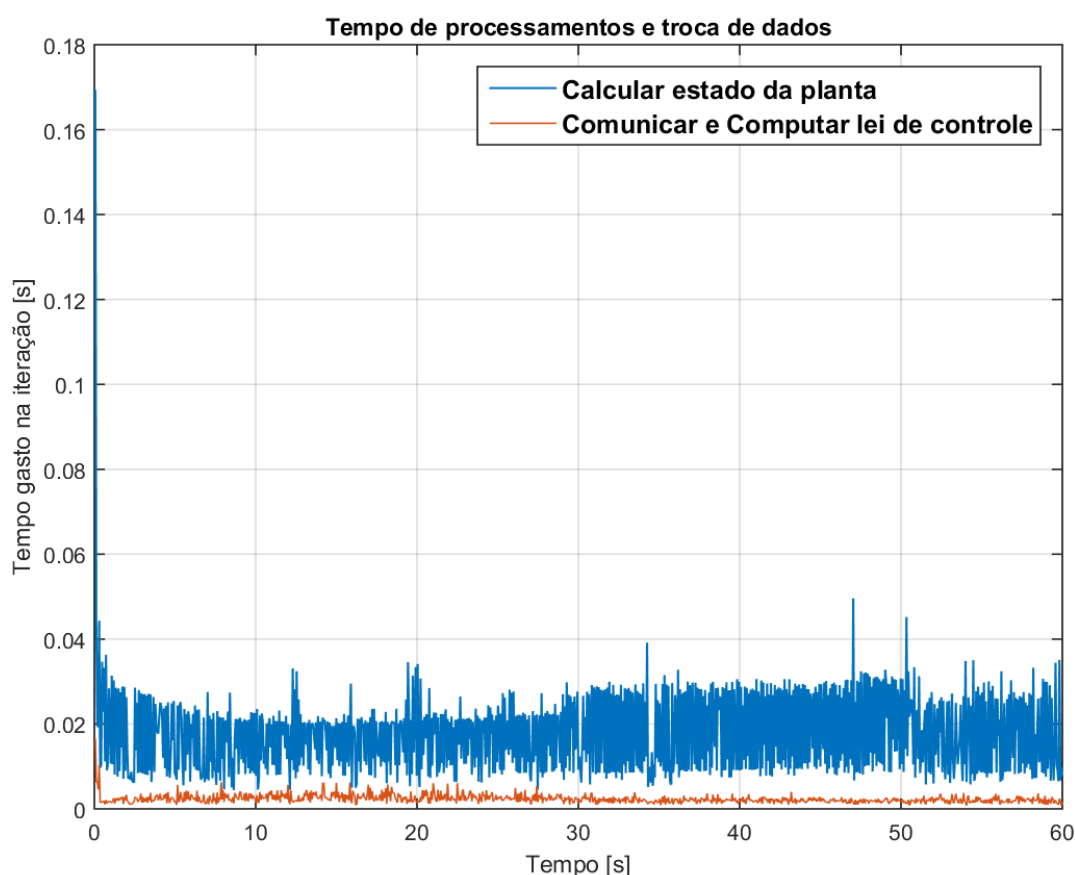


Figura 6.8 – Tempo necessário para calcular estado da planta (solucionar ODE) e para comunicar com microcontrolador e computar lei de controle, submetida a entrada degrau unitário, em função do tempo.

Destaca-se que o tempo gasto para solucionar a ODE é função dos recursos computacionais do computador onde se realizar este procedimento. Portanto, em computadores com maiores recursos o tempo necessário para a solução seria inferior ao desta simulação.

Por fim, são exibidas mais informações sobre o tempo de processamento e troca de dados por meio da Fig. 6.9 onde são mostrados os tempos durante os intervalos 40 s e 42 s da

simulação.

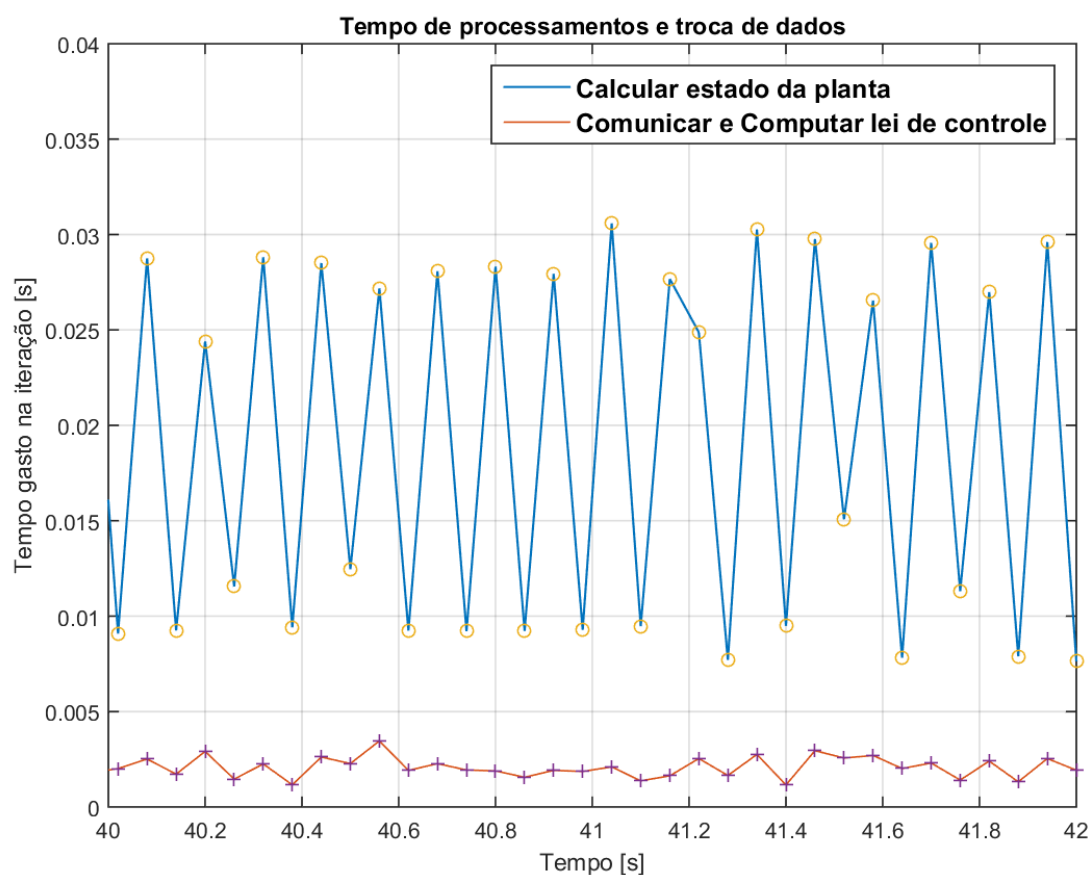


Figura 6.9 – Tempo necessário para calcular estado da planta (solucionar ODE) e para comunicar com microcontrolador e computar lei de controle, submetida a entrada degrau unitário, em função do tempo, com mais detalhes.

Como consideração final, deve-se dizer que a pausa realizada no cliente (MATLAB®), resulta na perda da discretização do sistema e compromete o tempo de amostragem. Desta forma, os resultados obtidos na simulação não refletem o exato comportamento que um VANT real apresentaria devido a esta pausa.

CAPÍTULO VII

CONCLUSÕES E TRABALHOS FUTUROS

7.1 Conclusões

Este trabalho expõe a implementação de um sistema *hardware-software* com finalidade de testar parâmetros e técnica de lei de controle. Para tal, é necessária plataforma *hardware* que contém estrutura mecânica do VANT, microcontrolador e sistema de propulsão. Também é indispensável o modelo dinâmico matemático não-linear que representa o VANT em questão implementado como rotina no *software* MATLAB®. Outra parte essencial são os componentes *software* implementados tanto no microcontrolador e em rotinas no MATLAB®. No domínio do microcontrolador foram desenvolvidos programas na linguagem de programação C e C++ para comandar o sistema de propulsão, implementar a lei de controle do tipo LQR e comunicar com a rotina cliente no MATLAB®. Já no campo referente ao MATLAB® foram implementadas a comunicação com o servidor (microcontrolador) e o modelo dinâmico do VANT.

A simulação *hardware-in-the-loop* realizada consiste nas interações entre microcontrolador e MATLAB® realizada utilizando plataformas *hardware* e *software* desenvolvidas pelo Laboratório de Aeronaves Autônomas e pelo autor deste trabalho. Dado o sinal de referência utilizado, um degrau unitário em cada eixo x_e , y_e e z_e o VANT convergiu. Também pode-se dizer que o tempo de amostragem utilizado neste projeto é adequado.

Então, podem ser feitas as seguintes conclusões: primeiramente a lei de controle LQR é eficiente no seu papel, uma vez que é responsável por o sistema convergir. Em segundo lugar,

a arquitetura de comunicação cumpriu seu papel pois é imprescindível a lei de controle receber o estado do VANT para calcular as saídas e estes dados devem ser recebidos na rotina do modelo dinâmico, visto que, são fundamentais para promoverem mudanças no modelo. Variações que podem ser visualizadas através dos dados obtidos referentes a posição e ângulos de Euler do CG do VANT. Em terceiro lugar, os recursos do sistema de enviar os parâmetros iniciais de controle através da rotina no MATLAB® e a capacidade do código fonte principal no microcontrolador entrar em *loop* para aguardar por solicitação de conexão de cliente se mostraram vantajosos uma vez que tornou o uso do sistema mais ágil. Destaca-se, também, que com os procedimentos deste sistema possibilitam a verificação do funcionamento dos atuadores (motores) antes de se implementar o controlador em um veículo que opera de forma autônoma. E por fim, o sistema apresenta relativa eficiência tendo em vista o tempo de processamento da lei de controle e o tempo necessário para a comunicação quando analisados em relação ao tempo de amostragem. Sendo assim, o sistema atende seu propósito.

7.2 Trabalhos Futuros

Apesar do sistema alcançar seu propósito, este pode ser melhorado e expandido. A fim de torná-lo melhor, pode-se otimizar os códigos do microcontrolador afim de torná-lo mais rápido utilizando-se mais ponteiros. Outra possibilidade seria implementar no sistema o uso de *frameworks* como por exemplo o *Robot Operating System*¹ (ROS) para que se compare performance, para utilizar recursos proporcionados pela *framework* e dentre outros.

Trabalhos adicionais que podem ser realizados são implementar outras técnicas de controle além da LQR, como por exemplo um controlador proporcional integrativo derivativo (PID). Outra opção é implementar sensoramento e realizar adequações que forem necessárias para utilizar o quadróptero desenvolvido pelo LAA no lugar do modelo dinâmico matemático e utilizar os resultados obtidos com o quadróptero a fim de comparar com os dados obtidos com o modelo. Outro trabalho extra seria incluir ruídos e o tempo computacional de atraso da simulação no modelo dinâmico. Outra possibilidade é utilizar o circuito integrado LM393 e medir a velocidade real das hélices e utilizá-las na simulação.

¹Disponível em: <<http://www.ros.org/>>. Acesso em: 3 jun. 2019.

CAPÍTULO VIII

REFERÊNCIAS BIBLIOGRÁFICAS

ARMADILLO. **Armadillo: C++ library for linear algebra & scientific computing**. [2016?].

Disponível em: <<http://arma.sourceforge.net/>>. Acesso em: 21 ago. 2018.

BRIANCODE. **Working with PWM on a BeagleBone Black**. 2015. Disponível em: <<https://briancode.wordpress.com/2015/01/06/working-with-pwm-on-a-beaglebone-black/>>. Acesso em: 07 fev. 2019.

BROUWER, A. **EXIT(3)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man3/exit.3.html>>. Acesso em: 21 mar. 2019.

BURNS, R. S. **Advanced Control Engineering**, Oxford: Plant a Tree, 2001.

CONROY, J.; KEHLENBECK, A.; HUMBERT, J. S.; NOTHWANG, W. **Characterization and Enhancement of Micro Brushless DC Motor Response**, 90831P, 10.1117/12.2051263, 2014.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos: Conceitos e Projeto**. 4. ed. São Paulo, SP: Bookman, 2007.

CPLUSPLUS.COM. **sprintf**. 2019. Disponível em: <<http://www.cplusplus.com/reference/cstdio/sprintf/>>. Acesso em: 13 fev. 2019.

DIE.NET. **close(2) - Linux man page**. [2010?]. Disponível em: <<https://linux.die.net/man/2/close>>. Acesso em: 21 jul. 2018.

DIE.NET. **close(3) - Linux man page**. [2010?] . Disponível em: <<https://linux.die.net/man/3/close>>. Acesso em: 21 jul. 2018.

DIE.NET. **htons(3) - Linux man page**. [2010?]. Disponível em: <<https://linux.die.net/man/3/htons>>. Acesso em: 21 jul. 2018.

DIE.NET. **send(2) - Linux man page**. [2010?]. Disponível em: <<https://linux.die.net/man/2/send>>. Acesso em: 21 jul. 2018.

DIE.NET. **send(3) - Linux man page**. [2010?]. Disponível em: <<https://linux.die.net/man/3/send>>. Acesso em: 21 jul. 2018.

ECKHARDT, D.; HAARDT, M.; JACKSON, I.; BANKS, G.; KERRISK, M. **OPEN(2)**. 2018. Disponível em: <<http://man7.org/linux/man-pages/man2/open.2.html>>. Acesso em: 21 jul. 2018.

ECKHARDT, D.; HAARDT, M.; JACKSON, I.; KERRISK, M. **WRITE(2)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man2/write.2.html>>. Acesso em: 20 mar. 2019.

FAITH, R. E. **BIND(2)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man2/bind.2.html>>. Acesso em: 15 mar. 2019.

FOGELBERG, J. **Navigation and Autonomous Control of a Hexacopter in Indoor En-**

viroments. Lund, Sweden: Lund University, 2013.

IBM® KNOWLEDGE CENTER. **Network byte order and host byte order**. 2018. Disponível em: <https://www.ibm.com/support/knowledgecenter/en/SSB27U_6.4.0/com.ibm.zvm.v640.kiml0/asonetw.htm>. Acesso em: 21 jul. 2018.

JORGE, L. A. de C.; INAMASU, R. Y. **Uso de Veículos Aéreos Não Tripulados (VANT) em Agricultura de Precisão**. In: BERNARDI, A. C. de C.; NAIME, J. de M.; RESENDE, A. V. de; BASSOI, L. H.; INAMASU, R. Y. (Ed.). *Agricultura de precisão: resultados de um novo olhar*. Brasília, DF: Embrapa, 2014. p. 109-134. Disponível em: <<http://ainfo.cnptia.embrapa.br/digital/bitstream/item/114264/1/CAP-8.pdf>>. Acesso em: 22 mai. 2019.

KLEEN, A. **IP(7)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man7/ip.7.html>>. Acesso em: 21 mar. 2019.

KLEEN, A.; WILCOX, M. **SOCKET(7)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man7/socket.7.html>>. Acesso em: 21 mar. 2019.

LAVRETSKY, E.; WISE, K. A. **Robust and Adaptive Control**, London: Springer, 2013.

LIANG, O. **ESC Firmware and Protocols Overview**. 2017. Disponível em: <<https://oscarliang.com/esc-firmware-protocols/>>. Acesso em: 25 abr. 2019.

LIANG, O. **What is Oneshot ESC Protocol – Active Braking**. 2015. Disponível em: <<https://oscarliang.com/oneshot125-esc-quadcopter-fpv/>>. Acesso em: 25 abr. 2019.

MACHINI, F. M. M. **Modelagem, Simulação e Controle de uma Aeronave Multirrotora**, Monografia, Universidade Federal de Uberlândia, Uberlândia - MG, Brasil, 2016.

MACHINI, F.; TARIFA, I. O.; NETO, R. M. F.; SANCHES, L. **Modeling, Control and Hardware-In-The-Loop Simulation of a Generic Tilting Multirotor**, International Council of the Aeronautical Sciences (ICAS), Belo Horizonte, Brazil, 2018.

MARQUES, F. M. M. **Modeling, Simulation and Control of a Generic Til-rotor Multi-copter**. Master Thesis, 99 pp, Federal University of Uberlândia, Uberlândia - MG, Brazil, 2018. Available in: <<https://repositorio.ufu.br/handle/123456789/22079>>.

MATHWORKS®. **ode15s**. 2019. Disponível em: <https://www.mathworks.com/help/matlab/ref/ode15s.html?searchHighlight=ode15s&s_tid=doc_srchttitle>. Acesso em: 09 jul. 2019.

MATHWORKS®. **tcpclient: Create TCP/IP client object to communicate over TCP/IP**. 2019. Disponível em: <<https://www.mathworks.com/help/matlab/ref/tcpclient.html>>. Acesso em: 20 jul. 2018.

MATHWORKS®. **Write data to remote host over TCP/IP**. 2019. Disponível em: <<https://www.mathworks.com/help/matlab/ref/tcpclient.write.html#buhajsp-2>>. Acesso em: 20 jul. 2018.

NATIONAL INSTRUMENTS. **Simulação Hardware-In-the-Loop (HIL)**. 2019. Disponível em: <<http://www.ni.com/pt-br/innovations/automotive/hardware-in-the-loop.html>>. Acesso em: 22 mai. 2019.

NATIONAL INSTRUMENTS. **What Is Hardware-in-the-Loop?**. 2019. Disponível em: <<http://www.ni.com/pt-br/innovations/white-papers/17/what-is-hardware-in-the-loop-.html>>. Acesso em: 22 mai. 2019.

PROGRAMAÇÃO PROGRESSIVA.NET. **Chamadas de Sistema - O que são, Para que servem e Exemplos de System Calls**. [2014?]. Disponível em: <<https://www.programaca>

oprogressiva.net/2014/09/0-que-sao-Chamadas-de-Sistema-para-que-servem-Exemplos-de-System-Calss.html>. Acesso em: 11 jan. 2019.

ROSKAM, J. **Airplane Flight Dynamics and Automatic Flight Controls**, DAR Corporation, Lawrence, KS, USA, 2001.

SANDERSON, C.; CURTIN, R. **A User-Friendly Hybrid Sparse Matrix Class in C++**. Lecture Notes in Computer Science (LNCS), Vol. 10931, pp. 422-430, 2018.

SANDERSON, C.; CURTIN, R. **Armadillo: a template-based C++ library for linear algebra**. Journal of Open Source Software, Vol. 1, pp. 26, 2016.

SINHA, A. **Socket Programming in C/C++**. [2017?]. Disponível em: <<https://www.geeksforgeeks.org/socket-programming-cc/>>. Acesso em: 25 jul. 2018.

SUIÇMEZ, E. C. **Trajectory Tracking of a Quadrotor Unmanned Aerial Vehicle (UAV) Via Attitude and Position Control**. Tubitak: Middle East Technical University, 2014.

TEACH ME MICROCONTROLLERS. **Beaglebone Black PWM on Ubuntu 16.04 Using Device Tree Overlay**. 2019. Disponível em: <<https://www.teachmemicro.com/beaglebone-black-pwm-ubuntu-device-tree/>>. Acesso em: 08 fev. 2019.

TEXAS INSTRUMENTS. **Linux Core PWM User's Guide**. 2019. Disponível em: <http://processors.wiki.ti.com/index.php/Linux_Core_PWM_User%27s_Guide>. Acesso em: 07 fev. 2019.

THE LINUX KERNEL ORGANIZATION, INC. **Pulse Width Modulation (PWM) interface**. 2014. Disponível em: <<https://www.kernel.org/doc/Documentation/pwm.txt>>. Acesso em: 07 fev. 2019.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. **ACCEPT(2)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man2/accept.2.html>>. Acesso em: 15 mar. 2019.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. **GETSOCKOPT(2)**. 2017. Disponível em: <<http://man7.org/linux/man-pages/man2/setsockopt.2.html>>. Acesso em: 21 mar. 2019.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA; KERRISK, M. **LISTEN(2)**. 2017. Disponível em: <<http://man7.org/linux/man-pages/man2/listen.2.html>>. Acesso em: 20 jul. 2018.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. **RECV(2)**. 2017. Disponível em: <<http://man7.org/linux/man-pages/man2/recv.2.html>>. Acesso em: 08 ago. 2018.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. **SOCKET(2)**. 2019. Disponível em: <<http://man7.org/linux/man-pages/man2/socket.2.html>>. Acesso em: 21 mar. 2019.

VALVANIS, K. P. **Advances in Unmanned Aerial Vehicles**, Ed. Springer, Dordrecht, Netherlands, 2007.

WIKIPÉDIA. **Electronic speed control**. 2019. Disponível em: <https://en.wikipedia.org/wiki/Electronic_speed_control>. Acesso em: 23 abr. 2019.

WIKIPÉDIA. **Secure Shell**. 2018. Disponível em: <https://pt.wikipedia.org/wiki/Secure_Shell>. Acesso em: 23 abr. 2019.

WIKIPÉDIA. **Soquete de rede**. 2019. Disponível em: <https://pt.wikipedia.org/wiki/Soquete_de_rede>. Acesso em: 25 jul. 2018.

CAPÍTULO IX

APÊNDICE

9.1 APÊNDICE A - Funções *Hexa*, *Forces* e *Moments*

Nesta seção são apresentadas as funções *Hexa*, *Forces* e *Moments*. Estas são implementadas de acordo com o modelo dinâmico apresentado no Capítulo de Fundamentação Teórica. Estas são responsáveis por calcular as forças e momentos atuantes na aeronave de acordo as velocidades angulares (*omega*) calculadas pela lei de controle do microcontrolador.

Hexa

```
1      function [dY] = Hexa(Y,param,omega,t)
2      global input_saida;
3      global referencia;
4      global time;
5      time(end+1)=t;
6      %% Reference signal
7      ref = zeros(3,1);
8      ref(1) = 1;
9      ref(2) = 1;
10     ref(3) = 1;
11     referencia(end+1,:) = ref';
```

```

12      % Rigid body parameters
13      m = param.m; % Multirrotor mass
14      J = param.J; % Multirrotor inertia matrix
15      Ixx=J(1,1); Ixy=J(1,2); Ixz=J(1,3);
16      Iyx=J(2,1); Iyy=J(2,2); Iyz=J(2,3);
17      Izx=J(3,1); Izy=J(3,2); Izz=J(3,3);
18
19      % State Variables
20      dY=zeros(12,1);
21      u=Y(1); v=Y(2); w=Y(3); P=Y(4); Q=Y(5); R=Y(6);
22      phi=Y(7); theta=Y(8); psi=Y(9);
23
24      %% Input calculation (Control Law)
25      input = omega;
26      input_saida(end+1,:)=input;
27      %% Dynamic equations for a 6dof rigid body in BCS
28      % Forces applied
29
30      F = Forces(Y,input,param,t);
31
32      Fx = F(1);
33      Fy = F(2);
34      Fz = F(3);
35
36      % Linear Momentum
37      dY(1) = Fx/m - Q*w + R*v;
38      dY(2) = Fy/m + P*w - R*u;
39      dY(3) = Fz/m - P*v + Q*u;
40
41      % Moments Calculation
42      Mom = Moments(param,input,Y,t);
43      L = Mom(1);
44      M = Mom(2);
45      N = Mom(3);
46
47      % Angular Momentum

```

```

48     dY(4) = (-Q*(-Izx*P-Izy*Q+Izz*R) + R*(-Iyx*P+Iyy*Q-Iyz*R) + L)/Ixx;
49     dY(5) = (P*(-Izx*P-Izy*Q+Izz*R) - R*(Ixx*P-Ixy*Q-Ixz*R) + M)/Iyy;
50     dY(6) = (-P*(-Ixy*P+Iyy*Q-Iyz*R) + Q*(Ixx*P-Ixy*Q-Ixz*R) + N)/Izz;
51
52
53     Mat = [1 sin(phi)*tan(theta) cos(phi)*tan(theta);
54            0 cos(phi) -sin(phi);
55            0 sin(phi)*sec(theta) cos(phi)*sec(theta)];
56     euler = Mat*[P;Q;R];
57     dY(7) = euler(1);
58     dY(8) = euler(2);
59     dY(9) = euler(3);
60
61
62     L1 = [cos(psi), sin(psi), 0;
63           -sin(psi), cos(psi), 0;
64           0, 0, 1];
65     L2 = [cos(theta), 0, -sin(theta);
66           0, 1, 0;
67           sin(theta), 0, cos(theta)];
68     L3 = [1, 0, 0;
69           0, cos(phi), sin(phi);
70           0, -sin(phi), cos(phi)];
71
72     LEB = L3*L2*L1;
73     V = LEB*[u;v;w];
74
75     dY(10) = V(1);
76     dY(11) = V(2);
77     dY(12) = V(3);
78
79     end

```

Forces

```

1      function F = Forces(Y,input,param,t)
2
3      %num_inp = param.n+param.tilt1+param.tilt2;
4      omg = input(1:param.n);
5      alpha = zeros(1,6);
6      beta = zeros(1,6);
7      % alpha = input(param.n+1:param.tilt1+param.n);
8      % beta = input(param.tilt1+param.n+1:num_inp);
9
10     u=Y(1); v=Y(2); w=Y(3);phi=Y(7); theta=Y(8); psi=Y(9);
11     % Rigid body parameters
12     n = param.n; % number of electric motors
13     m = param.m; % Multirrotor mass
14     kd = param.kd; % drag coefficient
15     k = param.k; %thrust coefficient
16     g = 9.81; % gravitational acceleration
17     gamma = param.gamma;
18     % Thrust Force
19     T_x = 0;
20     T_y = 0;
21     T_z = 0;
22     T = 0;
23     for i = 1:n
24         T_x = (sin(gamma(i))*sin(beta(i)) + ...
25             cos(gamma(i))*cos(beta(i))*sin(alpha(i)))*k*omg(i)^2 + ...
26         T_x;
27         T_y = (-cos(gamma(i))*sin(beta(i)) + ...
28             sin(gamma(i))*sin(alpha(i))*cos(beta(i)))*k*omg(i)^2 + ...
29         T_y;
30         T_z = +cos(beta(i))*cos(alpha(i))*k*omg(i)^2 + T_z;
31         T = k*omg(i)^2+T;
32     end
33     Fx = sin(theta)*m*g - kd*u + T_x;

```

```

34     Fy = -sin(phi)*cos(theta)*m*g - kd*v + T_y;
35     Fz = -cos(phi)*cos(theta)*m*g - kd*w + T_z;
36
37
38     F = [Fx,Fy,Fz]';
39
40     end

```

Moments

```

1     function Mom = Moments(param,input,Y,t)
2
3     % Rigid body parameters
4
5     n = param.n; % number of electric motors
6
7     k = param.k; %thrust coefficient
8
9     l = param.l; %multirrotor arm length
10
11    gamma = param.gamma; % multirrotor arm angle wrt x axis
12
13    JJm = param.JJm; % propeller moment of inertia
14
15    b = param.b; %propeller drag coefficient
16
17    zcg = param.zcg; % Center of gravity z position BCS
18
19
20    %num_inp = n+param.tilt1+param.tilt2;
21
22
23    omg = input(1:n);
24    alpha = zeros(1,6);
25    beta = zeros(1,6);
26
27    % alpha = input(n+1:param.tilt1+n);
28    % beta = input(param.tilt1+n+1:num_inp);
29
30    beta_d=[0,0,0,0,0,0];
31    alpha_d=[0,0,0,0,0,0];
32
33    % State Variables
34
35    P=Y(4); Q=Y(5); R=Y(6);
36
37
38    % Torques
39

```

```

25     L = 0;
26     M = 0;
27     N = 0;
28
29     for i = 1:n
30         % due to thrust
31         tau_x = ...
            l(i)*sin(gamma(i))*cos(beta(i))*cos(alpha(i))*k*omg(i)^2 + ...
            zcg(i)*sin(beta(i))*k*omg(i)^2;
32         tau_y = ...
            -l(i)*cos(gamma(i))*cos(beta(i))*cos(alpha(i))*k*omg(i)^2 + ...
            zcg(i)*cos(beta(i))*sin(alpha(i))*k*omg(i)^2;
33         tau_z = -l(i)*cos(gamma(i))*sin(beta(i))*k*omg(i)^2 - ...
            l(i)*sin(gamma(i))*cos(beta(i))*sin(alpha(i))*k*omg(i)^2;
34
35         % gyroscopic effect
36
37         Gy_x = (alpha_d(i) + Q) * JJm(i)*omg(i);
38         Gy_y = (-cos(alpha(i))*beta_d(i) + P)*JJm(i)*omg(i);
39         Gy_z = 0;
40
41         % Fan Torques
42         Ft_x = (-cos(beta(i))*sin(alpha(i))*b*omg(i)^2)*(-1)^i;
43         Ft_y = (sin(beta(i))*b*omg(i)^2)*(-1)^i;
44         Ft_z = (-cos(beta(i))*cos(alpha(i))*b*omg(i)^2)*(-1)^i;
45
46         L = tau_x + Gy_x + Ft_x + L;
47         M = tau_y + Gy_y + Ft_y + M;
48         N = tau_z + Gy_z + Ft_z + N;
49     end
50
51
52     Mom = [L,M,N]';
53
54     end

```